

# Manycore processing of repeated range queries over massive moving objects observations

Francesco Lettich <sup>\*1</sup>, Salvatore Orlando <sup>†1</sup>, Claudio Silvestri <sup>‡1</sup>, and Christian Jensen <sup>§2</sup>

<sup>1</sup>Dipartimento di Scienze Ambientali, Informatica e Statistica, Università Ca' Foscari,  
Via Torino 155, Venezia, Italy

<sup>2</sup>Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300,  
DK-9220 Aalborg Ø, Denmark

November 13, 2014

## Abstract

The ability to timely process significant amounts of continuously updated spatial data is mandatory for an increasing number of applications. Parallelism enables such applications to face this data-intensive challenge and allows the devised systems to feature low latency and high scalability. In this paper we focus on a specific data-intensive problem, concerning the repeated processing of huge amounts of range queries over massive sets of moving objects, where the spatial extents of queries and objects are continuously modified over time. To tackle this problem and significantly accelerate query processing we devise a hybrid CPU/GPU pipeline that compresses data output and save query processing work. The devised system relies on an ad-hoc spatial index leading to a problem decomposition that results in a set of independent data-parallel tasks. The index is based on a point-region quadtree space decomposition and allows to tackle effectively a broad range of spatial object distributions, even those very skewed. Also, to deal with the architectural peculiarities and limitations of the GPUs, we adopt non-trivial GPU data structures that avoid the need of locked memory accesses and favour coalesced memory accesses, thus enhancing the overall memory throughput. To the best of our knowledge this is the first work that exploits GPUs to efficiently solve repeated range queries over massive sets of continuously moving objects, characterized by highly skewed spatial distributions. In comparison with state-of-the-art CPU-based implementations, our method highlights significant speedups in the order of 14x-20x, depending on the datasets, even when considering very cheap GPUs.

## 1 Introduction

An increasing number of applications need to process massive spatial workloads. Specifically, we consider applications in settings where spatial data is continuously produced over time and needs to be processed rapidly, e.g., scenarios involving mobile device infrastructures, Massively Multiplayer Online Games (MMOG), behavioural simulations where the behaviours of agents may affect other agents within a given range, and so on. In these applications, very large populations of continuously *moving objects* frequently update their positions and issue *range queries* in order to look for other objects within their interaction area. The resulting massive workloads pose new challenges to data management techniques.

Figure 1 shows an instance of this setting with three objects, namely  $o_1$ ,  $o_2$ , and  $o_3$ . Object positions are represented as circles with object identifiers inside, while updates are depicted as arrows (labeled  $u_1, u_2, u_3$ ) connecting previous positions (represented by gray circles) with current positions. Range

---

<sup>\*</sup>lettich@dais.unive.it

<sup>†</sup>orlando@unive.it

<sup>‡</sup>silvestri@unive.it

<sup>§</sup>csj@cs.aau.dk

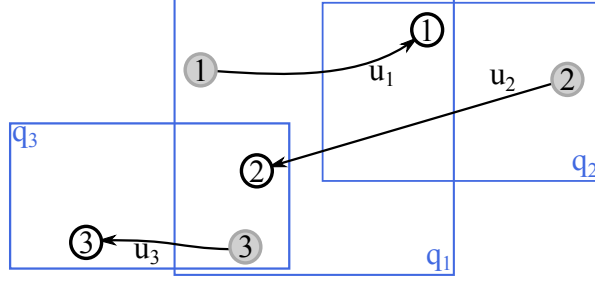


Figure 1: Moving objects, issued queries  $q$  and position updates  $u$ .

queries are shown as rectangles and labeled as  $q_1, q_2, q_3$ . The result set of query  $q_3$ , when executed after  $u_2$ , is  $\{o_2, o_3\}$ .

To enable parallel processing and optimizations, and thus manage the targeted workloads in a scalable manner, we discretize the time in intervals (*ticks*), assign location updates and queries to the ticks in which they occur, and process the updates and queries in the resulting batches such that the query results are reported after the end of each tick. This approach has the effect of replacing the processing of a large number of independent and asynchronous queries with the *iterated processing of spatial joins* between the last known positions of all the moving objects at the end of a tick and the queries issued during the tick. In other words we trade (slightly) delayed processing of queries for increased throughput, and therefore care is needed to ensure acceptable delays.

To achieve high performance and scalability we exploit a platform encompassing an off-the-shelf general-purpose microprocessor (CPU) coupled with a *Graphics Processing Unit* (GPU), which is a highly parallel *manycore* architecture featuring hundreds of processing cores [1, 2]. To benefit from GPUs exploitation, limitations and peculiarities of these architectures must be carefully taken into account. Specifically, individual GPU cores are slower than those of a typical CPU, while some memory access patterns may cause serious performance degradation due to contention and serialization of memory accesses. Effective query processing techniques must address these limitations: multiple cores must work together to efficiently process queries in parallel for most of the time, and must coordinate their activities to ensure high memory bandwidth.

With these goals in mind, we present the *Quadtree* method (hereinafter referred as QUAD), whose design allows an easy and elegant parallelization. The key idea behind the method is to partition the problem space using a *point-region quadtree* inducing a regular grid; in turn, the cells of the grid represent a set of independently solvable problems, each one associated with a *data-parallel task* runnable on a GPU streaming multiprocessor, also capturing and adapting to possibly skewed spatial distributions. This strategy allows us to obtain a quasi uniform distribution of the workloads among coarse-grained tasks – each task corresponding to a single cell of the index – with the aim of improving the overall efficiency of the system and maximizing the performance. In order to demonstrate the importance of this aspect we also introduce a baseline spatial index, whose space decomposition relies on the usage of a *uniform grid*. We call this method *Uniform Grid* (hereinafter referred as UG).

Both QUAD and UG preprocess in parallel the data and store consecutively object and queries falling in the same grid cell, thus optimizing memory accesses. Further, to avoid the use of blocking writes and to ensure high throughput, both methods compute the query results by means of a two phase-approach using a particular *bitmap* intermediate representation previously introduced by the authors [3].

To the best of our knowledge this is the first work that exploits the GPUs to efficiently solve repeated range queries over massive moving objects observations, having care to tackle effectively skewed spatial distributions as well. There are few existing works that use the GPUs for spatial query processing, but they consider substantially different problems, as detailed in Section 8.

In a previous work [3] we already introduced the general framework for repeatedly computing on GPU sets of range queries on streams of objects, but we limited ourselves to unrealistic uniform spatial distributions and to a spatial index relying on a simple uniform grid whose coarseness directly depends on the size of the queries. In this sense, this work aims to extend and improve the previous work by introducing a cleverer space index and a novel set of optimizations.

The main contributions of this work can be summarized as follows:

- we define a hybrid CPU-GPU pipeline to process batches of range queries, which effectively exploit the GPU computational power while taking care of its architectural features and limitations. Thanks to its flexibility, the pipeline can be adapted to different spatial indices.
- we introduce a set data structures which allow the pipeline to perform operations that concurrently write interlaced lists of results, using coalesced memory accesses that avoid race conditions.
- while we adopt the usual query splitting approach to make the data-parallel tasks induced by the indices completely independent, we take advantage of subquery areas that *completely cover* index cells to *save work* during the query processing (in terms of amounts of containment tests performed), and *compress* the information the GPU has to send back to the CPU when notifying the query results. This optimization is particularly important for facing one of the main data management challenges of our problem, related to the size of the aggregate output returned at each tick: for example, when the percentage of objects that issue a range query is 100%, this size is quadratic in the number of moving objects [3].
- we carry out an extensive set of experiments to compare QUAD with UG. The structural regularity and simplicity characterizing the uniform grids, onto which UG relies, is a well-known feature that fits very well the GPUs characteristics, but have structural limitations - mainly related to the inability to fully capture the skewness possibly characterizing the spatial data - which may seriously hinder the performances.

On the other hand, QUAD is able to automatically adapt to very skewed spatial object distributions, assuring very good performances for a broad range of spatial distributions. We demonstrate this claim by comparing QUAD with UG. We also compare QUAD with the state-of-the-art (for what relates to the problem considered) sequential CPU algorithm, namely the *Synchronous Traversal* algorithm [4, 5].

The paper is structured as follows: in Section 2 we proceed describing the problem setting and state the problem addressed. In Section 3 we cover the architectural specifics of the GPUs as well as the general constraints to be considered when designing hybrid CPU/GPU processing pipelines coupled with proper data structures. In Section 4 we present the spatial indices used by QUAD and UG to partition the workload, while in Section 5 we extensively detail the pipeline as well as the customizations needed by QUAD and UG. Sections 6 and 7 present an extensive set of experimental studies which show (i) the benefits coming from the usage of the proposed range query processing pipeline and data-structures, (ii) how QUAD spatial indexing is better with respect to the one used by UG and (iii) how QUAD outperforms the state-of-the-art sequential CPU competitor as well as outperform, or being on par with, UG. Finally, in Section 8 we cover the related work while Section 9 gives the conclusions.

## 2 Problem setting and statement

In this section we provide definitions that capture the problem setting and the problem we aim to solve.

### 2.1 Problem setting

We consider a set of points  $O = \{o_1, \dots, o_n\}$  moving in two-dimensional Euclidean space  $\mathbb{R}^2$ , where the position of object  $o_i$  is given by the function  $pos_i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^2$  mapping time instants into spatial positions.

These points model objects that issue position updates and range queries as they move. Let  $\mathcal{P}_i = \langle p_i^{t_0}, \dots, p_i^{t_k}, \dots \rangle$ ,  $t_j < t_{j+1}$ , be the time-ordered sequence position updates issued by  $o_i$ , where  $p_i^{t_j} = pos_i(t_j)$  is a position update. A range query issued by object  $o_i$  at time  $t$  is denoted by  $q_i^t = (x^a, x^b, y^a, y^b)$ , where  $(x^a, y^a)$  and  $(x^b, y^b)$  are the lower left and upper right corners of a rectangle. Thus,  $\mathcal{Q}_i = \langle q_i^{t_0}, \dots, q_i^{t_k}, \dots \rangle$ ,  $t_j < t_{j+1}$ , is the time-ordered sequence of queries issued by  $o_i$ .

Given the above, the most recently known position of  $o_i$  before time  $t$ ,  $t \geq t_0$ , is denoted as  $\hat{p}_i^t$  and defined as follows:

$$\hat{p}_i^t = p_i^{t_k} \in \mathcal{P}_i \text{ if } t_k < t \leq t_{k+1}$$

Similarly, the most recent query issued by  $o_i$  before time  $t$ ,  $t \geq t_0$ , is  $\hat{q}_i^t$ :

$$\hat{q}_i^t = q_i^{t_k} \in \mathcal{Q}_i \text{ if } t_k < t \leq t_{k+1}$$

We assume that the processing of a query can be delayed to a certain extent in order to optimize the overall system throughput. We process queries using the most up-to-date information available.

**Definition 1 (Result set of a range query)** *The result of query  $q_i^t$  when computed at time  $t'$ ,  $t_0 \leq t \leq t'$ , is denoted by  $res(q_i^t, t')$  and is defined as follows.*

$$res(q_i^t, t') = \{o_j \in O \mid \hat{p}_j^{t'} \in_s q_i^t\},$$

where  $\hat{p}_j^{t'} \in_s q_i^t$  denotes that  $\hat{p}_j^{t'} = (x, y)$  belongs to the query rectangle  $q_i^t$ , i.e.,  $x^a \leq x \leq x^b$  and  $y^a \leq y \leq y^b$ .

Assuming that updates  $u_1, \dots, u_3$  in Figure 1 are the most recent ones before  $t'$ , we have  $res(q_3^t, t') = \{o_2, o_3\}$ , which includes also object  $o_3$  that issued the query.

## 2.2 Batch processing

To obtain high throughput when facing massive workloads due to frequent updates and queries issued by huge populations of moving objects, we quantize time into *ticks* (time intervals) with the objective of processing updates and queries in batches on a per-tick basis. Assuming that the initial time is 0 and the tick duration is  $\Delta t$ , the  $k$ -th time tick  $\tau_k$  is the time interval  $[k \cdot \Delta t, (k+1) \cdot \Delta t)$ . Specifically, we aim to collect object position updates and queries that arrive during a tick, and process them at the end of the tick. If an object submits more than one update and query during a time tick, only the most recent ones are processed.

Let  $P^{\tau_k} = \{\hat{p}_1^{(k+1) \cdot \Delta t}, \dots, \hat{p}_n^{(k+1) \cdot \Delta t}\}$  be the last known positions of all objects at the beginning of the  $(k+1)$ -th tick, and  $Q^{\tau_k} = \{q_1^{\tau_k}, \dots, q_n^{\tau_k}\}$  be the most recent queries issued during the  $k$ -th tick, where:

$$q_i^{\tau_k} = \begin{cases} \hat{q}_i^{(k+1) \cdot \Delta t} & \text{If object } o_i \text{ issues any query during the } k\text{-th tick.} \\ \perp & \text{Otherwise.} \end{cases}$$

Note that if object  $o_i$  does not issue any query during the tick, then  $q_i^{\tau_k} = \perp$ .

Figure 2 captures the temporal aspects of the previous example. The timeline is partitioned into ticks  $\tau_1, \tau_2, \dots$  of duration  $\Delta t$ . Objects issue updates and queries independently and asynchronously. For

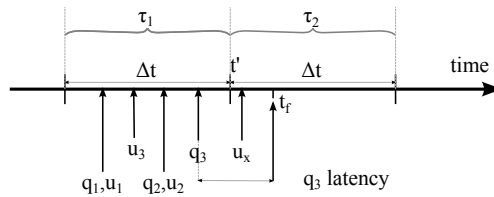


Figure 2: Timeline

example object  $o_3$  sends a query and an update separately. Incoming updates and queries are batched based on the ticks. For example, update  $u_x$  belongs to the batch of  $\tau_2$ . The batches are processed at the beginning of the next tick. Thus, at time  $t'$  (at the beginning of  $\tau_2$ ) we start processing all updates and queries arrived during  $\tau_1$ . We complete the processing of the batch, thus making available the query results, at time  $t_f$ , hopefully before the end of  $\tau_2$ .

## 2.3 Query semantics

The procedure for computing queries as described above ensures serializable query processing and implements the timeslice query semantics, where query results are consistent with the database state at a

given time, usually when we start processing the query. This is a popular choice in traditional databases and is commonly adopted in literature [6, 7, 8, 9].

For example, in Figure 2, the computation results are returned at time  $t_f$  for the first batch. Since the object update  $u_x$  occurs at time  $t_u$ ,  $t_u > t'$ , i.e., after we start processing  $q_3$ ,  $u_x$  is not considered even if it arrives before the results are returned. In this way, the query result is consistent with the database state at time  $t'$ , when we start processing query  $q_3$ .

## 2.4 Quality of Service - Query latency

On the one hand, the processing of updates and queries on large batches can be expected to improve system throughput. On the other hand, we assume that some applications, e.g., MMOG applications, are sensitive to the delays with which query results are returned. Thus, it is important to be able to assess the *latency* of query processing, which is affected by the number of queries and updates that arrive during a tick, the tick duration, and the computational capabilities of the system.

**Definition 2 (Latency, Queueing, Computation time)** Assume that the processing of query  $q_i^t$ , issued at time  $t$ , starts at time  $t'$  and completes at time  $t_f$ . We define the following durations:  $\text{latency\_time}(q_i^t) = t_f - t$ ,  $\text{queueing\_time}(q_i^t) = t' - t$ , and  $\text{processing\_time}(q_i^t) = t_f - t'$ , so that  $\text{latency\_time}(q_i^t) = \text{queueing\_time}(q_i^t) + \text{processing\_time}(q_i^t)$ .

We can now generalize the concept of latency to all the queries issued during a tick, all executed in batch at the beginning of the next tick.

**Definition 3 (Tick latency)** The latency of the queries  $Q^{\tau_k}$  arrived during the  $k$ -th tick is defined as follows:

$$\text{Tick\_Latency}(Q^{\tau_k}) = \max_{i \in \{1, \dots, n\}, q_i^{\tau_k} \neq \perp} \text{latency\_time}(q_i^{\tau_k})$$

Given an application-dependent maximum latency threshold  $\lambda$ , a system satisfies the application's *QoS Latency Requirement* if, for each tick  $k$ ,  $\text{Tick\_Latency}(Q^{\tau_k}) \leq \lambda$ .

The tick duration  $\Delta t$  should be chosen such that even queries issued at the beginning of a tick are answered within time duration  $\lambda$ . Since query processing is delayed till the beginning of the next tick, the *worst-case latency* for a query  $q_i^t$  takes place when  $q_i^t$  is issued at the beginning of a tick: in this case, the latency is the sum of  $\Delta t = \text{queueing\_time}(q_i^t)$  and  $\text{processing\_time}(q_i^t)$ . The following lemma states a simple, sufficient criteria to select  $\Delta t$  or to verify whether a given execution time satisfies the latency requirement.

**Lemma 1** Let  $\Delta t_{exe}^k$  be the time to process all queries in the  $k$ -th batch. Given a tick duration  $\Delta t$  and a latency requirement  $\lambda$ , then if  $\Delta t + \Delta t_{exe}^k \leq \lambda$ , the execution satisfies the latency requirements, i.e.,  $\text{Tick\_Latency}(Q^{\tau_k}) \leq \lambda$ .

From the above we can derive the following sufficient condition for  $\text{Tick\_Latency}(Q^{\tau_k}) \leq \lambda$ :

$$\Delta t > \lambda - \Delta t \geq \Delta t_{exe}^k. \quad (1)$$

Above, we have that  $\Delta t > \lambda - \Delta t$  because the processing of the queries accumulated during a tick is assumed to be completed before the end of the next tick.

The computational capabilities of a system influences the choice of the tick duration and the fulfillment of the latency requirement in Lemma 1.

**Lemma 2** Let  $\beta$  be the system bandwidth, expressed in terms of the number of queries processed per time unit, and let  $Q_{max}$  be the maximum number of queries that can occur during a tick. Then a sufficient condition for the system to meet the QoS latency requirement (based on threshold  $\lambda$ ) is:

$$\beta \geq \frac{Q_{max}}{\lambda - \Delta t} \quad (2)$$

**Proof 1** According to Equation 1, in order to respect the timeliness, we have to process all queries in a time  $\Delta t_{exe}^k$  such that  $\Delta t_{exe}^k \leq \lambda - \Delta t < \Delta t$ . Given the bandwidth  $\beta$ , the maximum execution time to process all queries of a tick is  $\frac{Q_{max}}{\beta}$ . Hence,  $\frac{Q_{max}}{\beta} \leq \lambda - \Delta t$  must hold, from which the lemma follows.

For a given latency requirement  $\lambda$ , if we increase the tick duration  $\Delta t$ , this increases  $Q_{max}$  and decreases  $\lambda - \Delta t$ . So, if we increase  $\Delta t$ , in order to satisfy Equation 2 we have to compute more queries in less time, and thus it may happen that bandwidth  $\beta$  becomes insufficient to support the requested workload respecting the given latency threshold, i.e.,  $\beta < \frac{Q_{max}}{\lambda - \Delta t}$ .

## 2.5 Problem statement

We can finally state the problem of computing repeated range queries over massive streams of moving objects observations, by discretizing the time in intervals (ticks), synchronizing query processing according to these ticks, and iteratively computing queries in batch mode.

Given (i) a set of  $n$  objects  $O$ , (ii) a partitioning of the time domain into ticks  $[\tau_k]_{k \in \mathbb{N}}$  of duration  $\Delta t$ , (iii) a query latency requirement  $\lambda$ , and (iv) a sequence of pairs  $[(P_{\tau_k}, Q_{\tau_k})]_{k \in \mathbb{N}}$ , where  $P_{\tau_k}$  is the up-to-date object positions at the end of  $\tau_k$ , and  $Q_{\tau_k}$  is the set of the last issued queries during  $\tau_k$ , we have that the **iterated batch processing** of queries  $Q_{\tau_k}$  over the corresponding  $P_{\tau_k}$ ,  $k \in \mathbb{N}$ , **yields**  $[R_{\tau_k}]_{k \in \mathbb{N}}$ , i.e., a **sequence of pairs**, each composed of a query and the list of the corresponding results:

$$R_{\tau_k} = \{(q_i^{\tau_k}, \text{res}(q_i^{\tau_k}, (k+1) \cdot \Delta t)) \mid q_i^{\tau_k} \neq \perp \wedge q_i^{\tau_k} \in Q_{\tau_k}\}.$$

The **processing time** of each batch of queries  $Q_{\tau_k}$  must be **upper bounded** as follows, to satisfy the query latency requirement  $\lambda$ :

$$\text{processing\_time}(Q_{\tau_k}) \leq (\lambda - \Delta t) < \Delta t$$

## 3 Graphics Processing Units

GPUs are based on massively parallel computing architectures that feature thousands of *cores* grouped in *streaming multiprocessors*<sup>1</sup> (hereinafter denoted by SMs for brevity) coupled with several gigabytes of high-bandwidth RAM. During the latest years these devices sparked a consistent interest due to their ability in performing general purpose computations which, together with the amount of available cores in their architectures, offer a potential for substantial performance gains when compared to the performance of traditional CPUs.

Due to the architecture of these devices, and depending on the problem considered, effectively exploiting their computational power is usually far from trivial. Specifically, each GPU processing core is slower than a typical CPU and has limitations on its access to device memory, resulting in potential contentions unless specific conditions are satisfied [10]. Moreover, GPU cores have to coordinate their actions, which is usually a complex issue considered their architectural organization.

Proper algorithms, designed with the architectures of the GPUs in mind, are needed in order to maximize the performances and obtain significant gains with respect to CPU-based algorithms, a goal which is not always possible to pursue [2] depending on the characteristics of the targeted problem.

In order to exploit effectively the computational power of a GPU, memory accesses should generally have high spatial locality in order to exploit GPUs demand-fetched caches as much as possible [11]. In addition, we have to ensure, whenever possible, that all the cores of an SM profit from memory block transfers, by forcing coalescing of parallel data transfers: this avoids serial memory accesses and consequent performance degradation due to a sub-optimal usage of memory hierarchies.

Moreover, GPUs feature several types of memories ranging from private thread registers and fast shared memory, which are both shared among the core groups of each SM, to global memory, which has a lower throughput but it is of significant size and represents the contact point with the CPU host. To achieve consistent performances a programmer has to be aware of this complex memory hierarchy by orchestrating and managing explicitly memory transfers between different memories.

Finally, workload partitioning is paramount when designing GPU algorithms since unbalances may create inactivity bubbles across the streaming multiprocessors and seriously cripple the performance.

A GPU consists of an array of  $n_{SM}$  multithreaded SMs, each with  $n_{core}$  cores, yielding a total number of  $n_{SM} \cdot n_{core}$  cores. Each SM is able to run *blocks* of *threads*, namely *data-parallel tasks*, with the threads

<sup>1</sup>For the purposes of this work we will use the NVIDIA CUDA terminology to refer to the GPUs architectural features and peculiarities, as well as to describe software targeted to GPUs, since CUDA represents the dominant framework in the context of general purpose computing on GPUs.



in a block running concurrently on the cores of an SM. Since a block typically has many more threads than the cores available in a single SM, only a subset of threads, called *warp*, can run in parallel at a given time instant. Each warp consists of  $sz_{warp}$  *synchronous, data parallel threads*, executed by an SM according to a SIMD paradigm [10, 12]. Due to this behavior, it is important to avoid branching inside the same block of threads. It is worth remarking that at warp level no synchronization mechanisms are needed to guarantee data dependencies among threads, thanks to the underlying scheduling. Finally, a function designed to be executed on GPU is called *kernel*.

### 3.1 Main algorithmic design issues

Considering the specificities of the problem described in Section 2.5, five main design issues shall drive the design of the hybrid CPU/GPU pipeline in charge of the query processing. First, we have to find a proper way to distribute the workload evenly among the GPU streaming multiprocessors, since unbalances typically create inactivity bubbles. Second, we need to avoid contention/serialization when accessing the GPU device memory, in order to favour spatial locality, thus properly taking advantage from the complex GPU memory hierarcies. Third, we should compress the data the GPU has to send back to the CPU during the query processing, since in our problem the output is typically much larger than the input. Fourth, *(iv)* expensive synchronization mechanisms among concurrent threads should be avoided, since these are typically very costly in terms of performance. Finally, for each pipeline task executed on the GPU the unit of parallelization (either objects or queries, or partitions of queries) should be carefully chosen according to the task specificities.

## 4 Spatial indexing and data structures

In this section we discuss the inspiring principles behind QUAD and UG, along with the architectural features of GPUs that impact on the spatial indices and data structures design.

When processing repeated range queries, the same procedure is repeated for each tick. Thus, for the sake of readability, hereinafter we omit the subscript that indicates the tick, and denote by  $P$ ,  $Q$ , and  $R$ , respectively, the up-to-date object positions, the non-obsolete queries, and the result set associated with a generic tick.

### 4.1 Design considerations

A brute-force approach for computing repeated range queries entails  $O(|P| \cdot |Q|)$  containment checks per tick. By using spatial indices it is possible to prune out consistent amounts of pairs of queries and object locations that do not intersect. However, when choosing or designing an appropriate index, we have to consider its pruning power along with its maintenance costs. For example, regular grid indices are generally reported to have low maintenance costs, and thus are suitable for update-intensive settings [13]. Another aspect is the number of cores and the memory hierarchy provided by the underlying computing platform. Given the same workload, different indices may be the best option for different platforms. With massively parallel platforms such as GPUs, the regularity characterizing spatial indices based on regular grids is attractive, as it enables fast and efficient parallel index updating and querying. Even if tree-based spatial indices are able to distribute objects evenly among the index cells (the tree leaves), we have to avoid navigating the tree, since this may severely hinder efficiency due to poor data locality when accessing the memory.

In a previous work [3] we devised a simple uniform grid-based spatial indexing used to partition the workload and prune out useless containment tests. In that work we chose to determine the size of the index grid cells on the basis of the query size: the rationale was to reduce the amount of index cells to be considered when processing each query. However, solutions based on uniform grids generally cannot cope efficiently with skewed spatial distributions. To solve this issue, in this paper we propose the QUAD method, which relies on a tree-based recursive spatial indexing, induced by point-region quadrees. To ensure an unbiased comparison between QUAD and uniform grid-based spatial indices, we also introduce the UG method as a baseline. UG relies on a simple uniform grid-based spatial index, without any a-priori constraint on the size of the grid cells. Also, UG integrates all the optimizations conceived for QUAD (such optimizations are detailed in Section 5.6).

## 4.2 Overview of the methods

In the following we give an overview of both UG and QUAD.

UG materializes at each tick a uniform grid over the minimum bounding rectangle enclosing the object positions. The only way UG can cope with data skewness is by changing the coarseness of the grid, targeting a coarseness tradeoff on the basis of the object densities in crowded areas and loosely populated ones.

QUAD still yields at each tick an index over the same bounding rectangle, but the index cells are of varied size as QUAD is able to dynamically tune their size according to local object densities. To this end QUAD utilizes a point-region quadtree, which entails a space partitioning that ensures a pretty balanced distributions of objects among the index cells even in presence of skewed data. Even if tree data structures are, in principle, difficult to manage on GPUs, the direct relationship between the quadtrees structural properties and the Morton codes [14, Ch. 2][15] open up to the possibility of implementing efficient massively parallel quadtree construction and lookup algorithms on GPUs[16].

Regardless of the index adopted, queries can be processed concurrently according to a *per-query parallelization*. More specifically, since both indices induce a partition of the space, where each disjoint space tile corresponds to a cell of either UG or QUAD, we virtually split queries according to the space partition, thus producing a *subquery* for each index cell a query intersects. Indeed, each subquery yields an *independent subtask*, which we process in parallel by only accessing the objects falling in the associated index cell. Note that this approach decreases the overall amount of containment checks, although the splitting yields more subtasks to process as the same query is processed several times, once for each relevant index cell.

It is worth pointing out that we can have subqueries whose areas entirely cover small index cells. This allows us to strongly optimize the computation: first, we can compress the output, since all the objects of these cells falls into the subquery areas; second, for the same reason we can avoid processing covering subqueries, thus saving computation time (see the *covering* queries optimization, Section 5.6).

Both UG and QUAD use an ad-hoc lock-free data structure based on bitmaps [3], to manage the result sets while they are produced on GPU. This design choice entails a further post-processing step needed to enumerate the final results contained in this data structure. To prove the merit of this choice we consider a more basic baseline, i.e., a variant of UG, namely the *Baseline Uniform Grid* (UG<sub>Baseline</sub>) method, which uses atomic operations to ensure the consistency of the result sets content.

## 4.3 Space partitioning and indexing

In the context of parallel query processing, there are two main reasons for partitioning and indexing the data according to a given space partitioning approach: the first one, also common to sequential query processing, is to avoid redundant computations and access to irrelevant data, while ensuring fast access to relevant information. The second reason, which is triggered by the ability to process data in parallel, is to ensure independent computations, avoid redundant work and balance the workload among the processing unit cores.

In the following we introduce the two space partitioning methods onto which UG and QUAD rely. Both methods aim to adaptively partition the Minimum Bounding Rectangle (MBR) containing all the object positions during any tick. We denote this MBR by  $\mathcal{G} = (x_a^{\mathcal{G}}, y_a^{\mathcal{G}}, x_b^{\mathcal{G}}, y_b^{\mathcal{G}})$ , where  $(x_a^{\mathcal{G}}, y_a^{\mathcal{G}})$  and  $(x_b^{\mathcal{G}}, y_b^{\mathcal{G}})$  represent the lower-left and the upper-right corners of the MBR. Aside from the specific ways through which the methods define the geometries and enumerate grid cells, both of them assign queries and object locations according to the same mapping functions (see Section 4.3.3).

### 4.3.1 Uniform grid-based partitioning

UG partitions the space by superimposing a uniform grid  $\mathcal{C}$ , whose cells are of equal size, over  $\mathcal{G}$ .

**Definition 4 (MBR partitioning into a uniform grid)**  $\mathcal{G}$  is partitioned according to a uniform grid  $\mathcal{C}$  of  $N \cdot M$  cells of width  $W$  and height  $H$  such that the cell  $c_{ij}$  covers the following region:

$$(x_a^{\mathcal{G}} + i \cdot W, \quad x_a^{\mathcal{G}} + (i + 1) \cdot W, \quad y_a^{\mathcal{G}} + j \cdot H, \quad y_a^{\mathcal{G}} + (j + 1) \cdot H).$$



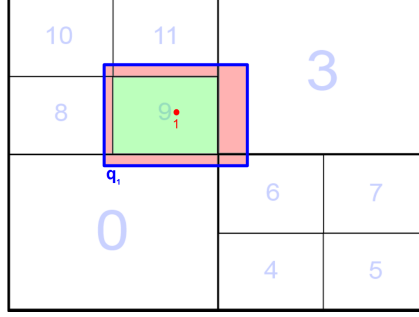


Figure 3: Simple mapping example over a quadtree-induced grid.

To ensure that the grid covers  $\mathcal{G}$ , constants  $N$ ,  $M$ ,  $W$ , and  $H$  are chosen so that  $x_a^{\mathcal{G}} + N \cdot W \geq x_b^{\mathcal{G}}$  and  $y_a^{\mathcal{G}} + M \cdot H \geq y_b^{\mathcal{G}}$  hold. We associate with each cell  $c \in \mathcal{C}$  an integer ID, which enforces a total order among the index cells, by preserving spatial locality.

#### 4.3.2 Quadtree-based partitioning

In the *quadtree based partitioning* case,  $\mathcal{G}$  is covered by a quadtree-induced grid  $\mathcal{C}$ , determined on the basis of the local densities of moving objects. In this case  $\mathcal{G}$  is therefore partitioned into a set of cells corresponding to the quadtree leaves.

**Definition 5 (MBR partitioning into a quadtree-induced regular grid)**  $\mathcal{G}$  is partitioned into a set of variably sized cells belonging to grid  $\mathcal{C}$ , induced by a point-region quadtree. Given a constant  $th_{quad}$ , denoting the maximum amount of objects allowed inside a single quadrant/cell of the final grid, we have that each cell of  $\mathcal{C}$  corresponds to a quadtree leaf, and contains an amount of object not greater than  $th_{quad}$ . We associate with each cell  $c \in \mathcal{C}$  an integer ID, which enforces a total order among the index cells, by preserving spatial locality.

#### 4.3.3 Mapping of moving objects and queries to space partitions

Given an index  $\mathcal{C}$  derived by QUAD or UG, we assign objects and queries to the index cells. Since the area of any query can intersect several cells of  $\mathcal{C}$ , this entails a partition of the area. We call this operation **query splitting**, which potentially yields a set of *subqueries* for each query. Finally, each subquery can be univocally assigned to a single index cell.

**Definition 6 (Mapping functions for object locations and subqueries)** Given the set of cells of a grid  $\mathcal{C}$ , we have two **mapping functions**  $f : P \rightarrow \mathcal{C}$  and  $g : Q \rightarrow 2^{\mathcal{C}}$  **map**. Function  $f$  maps each object location  $p \in P$  to the cell  $f(p)$  that contains  $p$ . Function  $g$  maps each query  $q \in Q$  to a set of cells  $g(q)$ , whose intersection with  $q$  is not empty. We use the term **subqueries** to denote the restrictions of a query  $q$  to each of these cells. Moreover, we call the operation performed by  $g$  **query splitting**. Finally, each subquery is classified as **intersecting** or **covering**, according to the fact that it partially/entirely covers the associated cell.

A simple example about how  $f$  and  $g$  operate is given in Figure 3:  $f$  maps object 1 to the cell having ID 9, while  $g$  splits query  $q_1$  (issued by object 1) over 7 different cells. Among these, six are *intersecting* ones (highlighted in pink, namely the subqueries intersecting cells with IDs 0, 3, 6, 8, 10, and 11) while one is a *covering* subquery (highlighted in green, covering the cell with ID 9).

### 4.4 Data structures

As it will be pointed out in Section 5, both QUAD and UG rely on a hybrid CPU/GPU processing pipeline, a pattern quite common in the context of General Purpose Computing on GPUs [17, 18]. Each stage of the pipeline performs a set of transformations on the data in order to produce a final output. To this end, the design of data structures should (i) allow data to be concurrently accessed with minimal use of atomic operations or barriers, thus avoiding locking related penalties; (ii) permit

the use of coalesced memory accesses, in order to maximize the memory throughput; (iii) exploit spatial locality, whenever possible, in order to maximize the benefits deriving from coalescing and caching. In the following subsections we introduce the relevant data structures used by our approach.

#### 4.4.1 Moving objects and queries data structures and their layout

Given a set of  $n$ -tuples representing a class of entities (in our context an object location or a query), the tuples elements are logically arranged by means of a *structure of vectors* (also known as *structure of streams* or *structure of arrays*) layout [19, Ch.33]. This layout groups a set of  $n$  vectors, each one representing a single element of the tuples, and aligns the vectors elements with respect to the entities they are associated with. An example of such arrangement is given in Figure 4.

The main benefits of this representation derive from the observation that it does not require complex pointer arithmetic, and it naturally makes possible to exploit coalesced memory accesses. Moreover, it is a representation commonly used in well established GPU algorithms, thus allowing an efficient interplay (and code reuse) between the operations making up the processing pipeline.

Consider that we aim at generating *independent tasks*, each one associated with a *subquery* and a specific *active cell*, i.e., a cell with at least one object and one subquery, where each task processes a single subquery over the objects of the associated cells (see Sections 5.4 and 5.5 for more details). In light of this, it is convenient to properly arrange the structures of vectors associated with objects and subqueries (each set has its own structure) in order to exploit data spatial locality and boost memory throughput. To this end, we have to arrange entities falling inside the same grid cell in contiguous memory locations (memory blocks). Therefore, first object locations and subqueries are sorted by the IDs of the associated index cells. As a side-note, we observe that the same originating query can be stored in several memory blocks, since function  $g$  potentially yields multiple intersecting cells for each query. Second, block boundaries are stored in a table, by distinguishing between the sub-blocks storing object locations and sub-blocks storing subqueries. This allow us to directly access the data belonging to any cell. The reader may refer to Section 5.3 for more details about the sorting operations performed in order to achieve such arrangements.

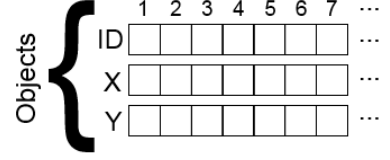


Figure 4: Example of a structure of vectors used on a set of objects described by 3-tuples.

#### 4.4.2 Intermediate bitmap representations of query result set

One of the main issues is related to the efficient collection of possibly huge sets of query results. According to the problem statement given in Section 2.5, the result of a single tick is described in terms of a set of pairs, each one consisting of an identifier associated with the object issuing a query and a set of identifiers related to the objects falling inside the query result set. Since query results are produced concurrently, contentions when writing them out could seriously cripple massive parallelism.

To avoid this issue we exploit and improve the two-phase approach we introduced in a previous work [3], which relies on two intermediate data structures based on a *bitmap* layout in order to eliminate the need of threads synchronizing mechanisms while maximizing the overall memory throughput and minimize the amount of space used to store intermediate results on GPU. Since these data structures are strongly tied to the design of the algorithms in charge of the aforementioned operations, we postpone their description to Sections 5.4 and 5.5.

## 5 Query processing pipeline

The core computation to process each set of range queries can be surely ascribed to the containment tests between objects locations and query areas. Considering the potential huge amount of containment tests and results each tick can yield, this apparently simple and straightforward operation is very expensive. In order to improve its efficiency we embed this computation in a pipeline of concatenated operations. The various stages of the pipeline prepare the spatial index for improving the efficiency of the containment

tests, compute the containment test outcomes in an intermediate format for efficiency reasons, and post-processes these results to produce the final query results.

In the following subsections we introduce the high-level pipeline, common to all methods. We also discuss the main design differences between UG, UG<sub>Baseline</sub> and QUAD in the implementation of each pipeline phase.

## 5.1 Pipeline description

The data entering the processing pipeline at the end of each tick are first processed to select the index parameters and build an empty index (phase 1, *index creation*). Then (phase 2, *moving object and query indexing*), objects and subqueries are mapped to index cells, and finally are sorted so that those contained in the same cell are stored in contiguous memory locations. The subsequent phase computes the containment tests between range queries and object locations (phase 3, *filtering with bitmap encoding*) producing an intermediate bit-encoded output which is structured to avoid contentions in memory access (issues (ii) and (iv) in Section 3.1). These intermediate results need a final post-processing phase to extract the final results (phase 4, *bitmap decoding*). Each phase takes advantage of a tight cooperation between the GPU and the CPU.

One of the key features of QUAD and UG is the ability to split the computation of each query among the space partitioning elements (cells) it intersects to reduce the total amount of containment tests. This entails the creation of a new set of *subqueries* originating from the query set  $Q$ . The distinction between UG (UG<sub>Baseline</sub>) and QUAD is related to the way they partition the space, that is, how a grid is materialized over the space and how objects locations and subqueries are mapped to grid cells. These aspects involve just the phases 1 and 2 of the pipeline, since the remaining ones directly use the cell identifiers associated with object locations and subqueries to determine which object locations and subqueries are relevant for a specific operation.

For this reason, in the following we describe phases 1 and 2 separately for UG (UG<sub>Baseline</sub>) and QUAD, while the remaining ones can be described regardless of the involved spatial index.

## 5.2 Index creation and indexing in UG and UG<sub>Baseline</sub>.

The performances of this method are significantly affected by the cells size used for  $\mathcal{C}$ . Choosing a suitable value is challenging since it depends on several factors, from the spatial distribution of data, to the opportunity of avoiding part of the computations thanks to optimizations that are triggered locally by grid and query based conditions (e.g., by exploiting the *covering* subqueries optimization described in Section 5.6).

In a previous work [3] we were able to optimally determine the cell size of a uniform grid index, by assuming unrealistic uniform spatial distributions of objects. Since the optimal granularity cannot be decided for any kind of dataset, but we need to still use uniform grid indexes as baselines for QUAD, we exploit an oracle to choose the grid coarseness for both UG and UG<sub>Baseline</sub>. In practice, we determine the optimal grid coarseness parameter, for each tick and any kind of dataset, by performing parameter sweeping, and finally selecting the parameters that are the most favorable to UG and UG<sub>Baseline</sub> in each comparison. Accordingly, the goal of the index creation phase for the baselines UG and UG<sub>Baseline</sub> is simply the ad-hoc choice of the best grid granularity, in order to maximize the performance of the subsequent phases.

**Index creation (UG/UG<sub>Baseline</sub>).** Since we already know the optimal grid cell size, the goal of the *index creation* phase in UG/UG<sub>Baseline</sub> is to determine the minimum rectangle  $\mathcal{G}$  that bounds all the objects (*MBR*). The computation of the MBR is based on a GPU parallel reduction operation over the set of object positions and queries yielding the minimum and maximum coordinates.

Once  $\mathcal{G}$  is set up, we use the cell size determined by the oracle to materialize an optimal uniform grid  $\mathcal{C}$  over  $\mathcal{G}$ , so that objects and queries can be indexed accordingly.

Each cell of  $\mathcal{C}$  is naturally associated with a pair  $(i, j)$ , identifying the row and the columns of each cell. However, we adopt a transformation of  $(i, j)$  into a uni-dimensional identifier *CellID*, derived from  $(i, j)$  by interleaving the binary representations of the two coordinates, thus obtaining the Morton code

$z(i, j)$ <sup>2</sup>.

**Moving objects and queries indexing (UG/UG<sub>Baseline</sub>).** Given an index  $\mathcal{C}$ , function  $f$  (Definition 6) maps a generic object location  $p \in P$  to a cell  $c \in \mathcal{C}$ . In UG/UG<sub>Baseline</sub> the function consists of a simple algebraic expression that determines grid coordinates (which indeed correspond to a unidimensional Morton code identifying the cell) from object locations. This is implemented on the GPU by applying function  $f$  in parallel to all elements of  $P$ , thus obtaining a vector whose elements represent cell identifiers corresponding to each object location.

Still on the basis of index  $\mathcal{C}$ , function  $g$  (Definition 6) maps a generic range query  $q \in Q$  to a set of cells in  $\mathcal{C}$ . The corners of each query  $q$  are mapped to grid coordinates, then a nested loop is used to enumerate the identifiers of cells intersected by the query. Since containment tests are superfluous for cells completely covered by  $q$ , the corresponding subqueries are marked as *covering* to enable the optimizations described in Sec. 5.6.

In our GPU implementation of  $g$ , each query  $q$  is processed by a GPU thread that produces a set of triples  $(queryID, cellID, coveringFlag)$ <sup>3</sup>, each one representing an intersecting (covering) subquery. To avoid output write contentions without resorting to blocks and synchronization, a two-pass approach is adopted: the first dry-run pass determines the amount of triples per query, while the second pass writes out the triples to the correct positions in the output vector by exploiting the information created during the first pass. During the second pass, each subquery is also classified according to the intersecting/covering dichotomy.

The overall complexity of this phase is equal to  $O(|P| + 2|Q| + |Q| + |\hat{Q}|) = O(|P| + 3|Q| + |\hat{Q}|)$ :  $|P|$  is due to the object locations indexing,  $2|Q|$  is due to the two-pass approach,  $|Q|$  is the cost to pay for the exclusive prefix sum performed between the first and the second pass needed to determine the subqueries locations in memory; finally,  $|\hat{Q}|$  is related to the subqueries written out during the second pass.

**Sorting (UG/UG<sub>Baseline</sub>).** Once object locations and subqueries are mapped to cells of  $\mathcal{C}$ , we sort them by the Morton codes of the cells, as illustrated in Figure 5. The goal is to store tuples mapped to the same index cell in contiguous memory locations, thus enhancing the *spatial locality* of each parallel block of threads working on subqueries and objects of a given active cell (i.e., a cell having at least one object) during the subsequent *filtering* and *decoding* phases (Sections 5.4 and 5.5 respectively).

Indeed, when sorting the subqueries we distinguish between covering and intersecting ones, in order to support the optimizations discussed in Section 5.6. In practice, we handle the covering queries in a different way, since the GPU does not need to process them: after the sorting operation, all intersecting subqueries, which need to be processed, are placed at the beginning of the subqueries structure of vectors.

Since the GPU sorting algorithm used throughout the pipeline will be the Radix Sort [18], the complexity of the sorting step is  $O(b \cdot (|P| + |\hat{Q}|)) \approx O(|P| + |\hat{Q}|)$ , where  $\hat{Q}$  denotes the subqueries set.

### 5.3 Index Creation and Indexing in QUAD

The key idea behind QUAD is to use a point-region (PR) quadtree as the backbone of its spatial index, exploiting the PR-quadtrees intrinsic ability to partition the space in differently sized parcels containing similar amounts of points.

**Index creation (QUAD).** The goal of this phase is to create a space partitioning  $\mathcal{C}$  over  $\mathcal{G}$ , according to Definition 5, where each cell of  $\mathcal{C}$  is a leaf PR-quadtree quadrant that does not contain more

		Unsorted							Sorted						
ID		0	6	2	3	4	5	1	2	1	0	3	4	5	6
z		1	3	0	2	2	2	1	0	1	1	2	2	2	3

Figure 5: A simple example of a GPU-based sorting, based on the structure of vectors representation, of 8 entities according to their Morton codes. The discontinuities among the codes (thicker lines) determine the set of entities belonging to each cell.

<sup>2</sup>To this end, we adopt an optimized bitwise algorithm.

<sup>3</sup>In practical terms, the *coveringFlag* can be properly embedded inside the integer representing *cellID*.

than  $th_{quad}$  objects. This property gives an upper bound to the containment tests computed by each GPU thread in charge of processing a query over all the objects falling in an index cell.

---

**Algorithm 1:** GPU-based PR-quadtrees construction

---

```

1 begin
2    $V_P \leftarrow GPUcalculateMortonHash(V_P, I_A, l_{max})$ 
3    $V_P \leftarrow GPUradixSort(V_P)$ 
4    $I_A \leftarrow \{[0, |P| - 1]\}$ 
5    $\mathcal{C} \leftarrow \emptyset$ 
6    $l \leftarrow 1$ 
7   repeat
8      $I \leftarrow GPUdetectQuadrants(V_P, I_A, l, l_{max})$ 
9      $(I_A, \mathcal{C}) \leftarrow CPUcheckQuadrants(th_{quad}, I, l, l_{max}, \mathcal{C})$ 
10     $l_{deep} \leftarrow l$ 
11     $l \leftarrow l + 1$ 
12  until  $(I_A \neq \emptyset) \wedge (l \leq l_{max})$ 
13   $z_{map} \leftarrow GPUbuildZMap(\mathcal{C}, l_{deep})$ 

```

---

We observe that even if a space partitioning is determined according to local object densities for a particular tick, it can be often reused for consecutive ticks when the spatial distribution does not change significantly.

Therefore we compute the spatial quadtree partitioning during the first tick, and repeat this partitioning if the objects spatial distribution change significantly, since this event might potentially hinder the performances by increasing the overall amount of containment tests to be computed per query.

The construction of the quadtree proceeds top-down in an iterative manner, starting from the 4 equally sized quadrants that partition  $\mathcal{G}$ , and then splitting iteratively each quadrant containing more than  $th_{quad}$  objects. The whole procedure is repeated level-wise, increasing the quadtree depth and splitting overpopulated quadrants if needed.

Algorithm 1 describes this iterative process. During the initial setup (lines 2 – 6), the function *GPUcalculateMortonHash* (line 2) computes the Morton codes  $z$  of all the objects stored in the structure of vectors  $V_P$  at the maximum quadtree level  $l_{max}$ . In practice, in this phase we consider a regular grid having  $2^{l_{max}} \times 2^{l_{max}}$  cells. Morton codes  $z$  are computed in the same way as done in the *UG/UG<sub>Baseline</sub>* case, starting from the index  $(i, j)$  of the regular grid where each object falls into. Subsequently,  $V_P$  is reordered by *GPUradixSort* (line 3) according to the Morton codes  $z$ . Note that, given the  $z$ -code at the maximum quadtree level  $l_{max}$ , we can determine the quadrant index  $z'$  of any object at any level  $l \leq l_{max}$  by simply truncating the binary representation of the Morton code  $z$  previously computed, which is equivalent to calculating  $z' = \frac{z}{4^{l_{max}-l}}$ . It is worth considering that the object order obtained by this sorting by  $z$  is invariant for any level  $l \leq l_{max}$  of the quadtree. In other words, thanks to this sorting and the structural properties of quadtrees, objects contained in any quadtree leaf are memorized contiguously in  $V_P$ .

Subsequently, the algorithm initializes several variables: the set  $\mathcal{C}$  of final leaves is initialized to  $\emptyset$ , the set  $I_A$ , containing the intervals of the quadrants to split, is initialized by inserting the interval related to the tree root, and, finally, the level  $l$  from which the iterative construction starts is set to  $l = 1$  (lines 4 – 6).

Then, the algorithm iteratively builds (line 7) the quadtree level by level. *GPUdetectQuadrants* (line 8) identifies the starting and ending positions (i.e., the intervals) of the  $l$ -level quadtree quadrants related to the  $(l - 1)$ -level quadrants added to  $I_A$  for splitting, and store such intervals in  $I$ . Then, *CPUcheckQuadrants* (line 9) determines which quadrants need further splitting at next level (their intervals are added to  $I_A$ ) and which quadrants represent final leaves (their identifiers are added to  $\mathcal{C}$ ). The process ends whenever no more quadrants need to be split (i.e.,  $I_A$  is empty) or the maximum possible quadtree level  $l_{max}$  is reached (line 12). In the latter case, all the quadrants found at level  $l_{max}$  are added to  $\mathcal{C}$ . We postpone the description of *GPUbuildZMap* (line 13) to a subsequent paragraph (see **Indexing moving objects and queries (QUAD)**).

Functions *GPUcalculateMortonHash*, *GPUradixSort* and *GPUdetectQuadrants* are entirely implemented on GPU. On the other hand, *CPUcheckQuadrants* is executed on the CPU side, since the amount of quadtree quadrants created at each level are typically orders of magnitude lower than  $|P|$ .

*Simple running example.* Let us consider the example reported in Figure 6, where  $l_{max} = 2$  and  $th_{quad} = 1$ . During the *Initialization* step, each object identified by an ID is associated with the Morton code  $z$  of the cell  $c \in \mathcal{C}_{l_{max}}$ , where  $\mathcal{C}_{l_{max}}$  denotes a uniform grid, associated with the deepest possible quadtree level  $l_{max}$  (see the “Initialization” grid on the left of the figure). The pairs  $(ID, z)$  are stored in a table (see the “Unsorted” table). Subsequently, pairs are sorted according to the second elements, i.e., the Morton codes (see the “Sorted” table). Next, the algorithm proceeds building the quadtree, starting from *Level 1*, creating iteratively new levels, until at least one quadrant requires to be split ( $I_A \neq \emptyset$ ) or  $l_{max}$  is reached.

At each level, the algorithm associates each object with a quadtree quadrant belonging to the currently considered level by computing the corresponding quadrant indices  $z'$ . In this regard see the “Iterations” table in Figure 6, where each row (after the second one) corresponds to an algorithm iteration working on a distinct level of the quadtree. On the right side of the same figure, we can also observe how the quadtree grows up at each iteration/level. More specifically, at each iteration the algorithm determines which quadrants need to be split. Objects falling in quadrants to split are re-assigned by computing the new quadrant indices  $z'$  (highlighted in red in the “Iterations” table). Quadrants that have not to be split are added to the set of  $\mathcal{C}$  cells. Objects belonging to the latter kind of quadrants (highlighted in green in the “Iterations” table) can be ignored during the successive iterations (the ignored cells are highlighted in grey in the “Iterations” table), since these already belong to quadtree leaves.

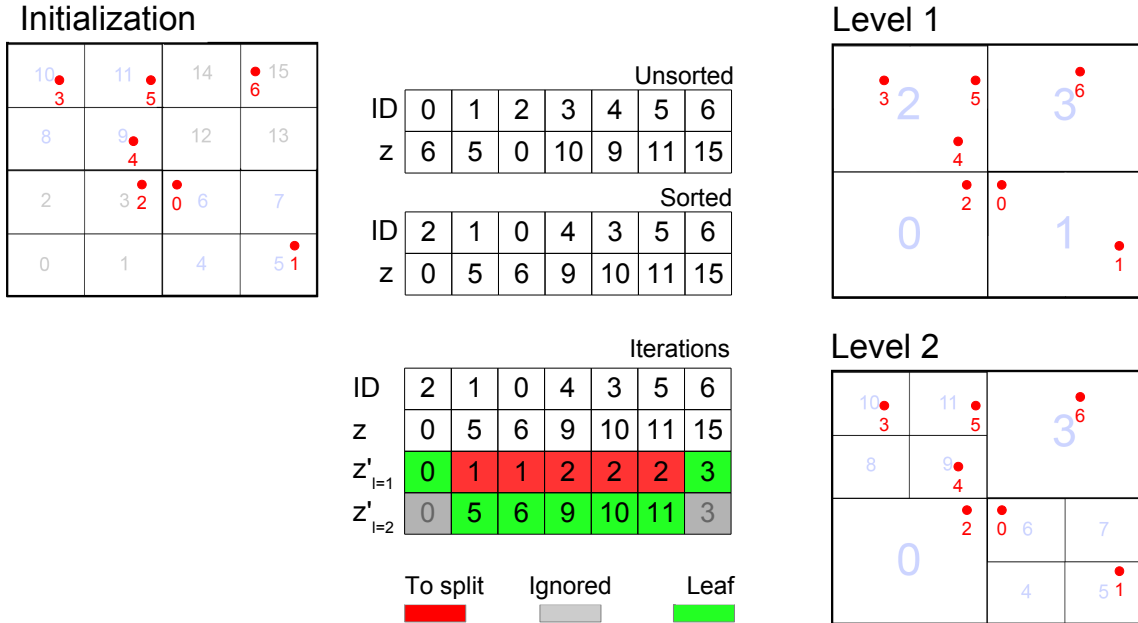


Figure 6: Example of quadtree construction with 7 objects,  $th_{quad} = 1$  and  $l_{max} = 2$ .

In the running example the algorithm stops at **Level 2**, since all the quadrants created at this level contain an amount of objects not greater than  $th_{quad}$ . Note that also the maximum quadtree level  $l_{max} = 2$  is reached for 8 leaves of 10.

*Complexity.* The computation of a single Morton code has a fixed cost determined by the number of bits used for coordinate representation; therefore,  $GPU_{calculateMortonHash}$  complexity is equivalent to  $O(|P|)$ .  $GPU_{radixSort}$  complexity is  $O(b \cdot |P|) \simeq O(|P|)$ , where  $b$  represents the base value during sorting ( $b \ll |P|$ ).  $GPU_{detectQuadrants}$  worst-case complexity is  $O(l_{max} \cdot |P| + 2 \sum_{l=0}^{l_{max}} 4^l)$ , where the first term is due to the amount of objects scanned in  $V_P$  at each iteration, while  $2 \cdot \sum_{l=0}^{l_{max}} 4^l = 2 \cdot \frac{1-4^{l_{max}+1}}{1-4}$  represents the maximum amount of starting and ending indices - which has to be written out in memory - related to the  $4^l$  quadtree quadrants at any level  $l$ . We observe that the amount of quadrants created at each level is orders of magnitude lower than  $|P|$ , hence the related computational overhead is negligible. As a consequence, the average complexity can be safely approximated to  $O(l_{max} \cdot |P| + 2 \sum_{l=0}^{l_{max}} 4^l) \simeq$



$O(l_{max} \cdot |P|)$ . *CPUcheckQuadrants* has a worst-case complexity equal to  $\sum_{l=0}^{l_{max}} 4^l = \frac{1-4^{l_{max}+1}}{1-4}$ . Again, its complexity is practically negligible according to the above considerations.

Summing up, the complexity of the iterative process is dictated by the number of objects processed and the depth  $l_{deep} \leq l_{max}$  reached in the quadtree construction, yielding  $O(l_{deep} \cdot |P|)$ . Since  $l_{deep}$  is usually a low constant, the overall complexity can be approximated to  $O(|P|)$ .

**Building a lookup table to map coordinates to cells (QUAD).** The usual approach for finding the quadtree leaf that corresponds to the coordinates of an object would consist in traversing the tree from the root, recursively choosing the relevant node until a leaf is reached. Unfortunately this approach entails repeated irregular memory accesses and a non predictable number of operations for each leaf search. The second issue, in particular, would cause branch divergence and potential sub-optimal occupancy of GPU cores.

For this reason we use a different approach, characterized by a slightly larger memory footprint. Let us suppose that the deepest level created in a quadtree  $\mathcal{C}$  is  $l_{deep}$ ,  $l_{deep} \leq l_{max}$ . Thus we virtually divide the space covered by  $\mathcal{C}$  according to a uniform squared grid composed of  $2^{l_{deep}} \times 2^{l_{deep}}$  cells, and denote it by  $\mathcal{C}^{l_{deep}}$ . In other words, we cover  $\mathcal{C}$  such that each quadtree leaf created at level  $l_{deep}$  corresponds exactly to a single cell in  $\mathcal{C}^{l_{deep}}$ . Thanks to the PR-quadtree properties, any quadtree leaf at a level  $l$ ,  $l \leq l_{deep}$ , corresponds to the union of  $4^{(l_{deep}-l)}$  contiguous cells of  $\mathcal{C}^{l_{deep}}$ . Therefore, a mapping between  $\mathcal{C}^{l_{deep}}$  cells and  $\mathcal{C}$  cells can be easily established by means of a lookup table  $z_{map}$ , which maps each  $\mathcal{C}^{l_{deep}}$  cell, identified by a pair  $(i, j)$ , to the  $\mathcal{C}$  cell containing it.

The idea behind this approach is exemplified in Figure 7. The example is derived from the one in Figure 6, and therefore  $l_{deep} = 2$ . Each  $\mathcal{C}$  cell (quadtree leaf) is identified by a pair  $(l, z)$  (an integer is indeed sufficient to store each pair), where  $l$  is the leaf level and  $z$  its Morton code at level  $l$ , whereas each cell in  $\mathcal{C}^{l_{deep}}$  is associated with the pair  $(l, z)$  identifying the cell of  $\mathcal{C}$  containing it. We can observe that 4 distinct  $\mathcal{C}^{l_{deep}}$  cells are mapped to the same  $\mathcal{C}$  cell  $(1, 0)$ , and other 4 distinct  $\mathcal{C}^{l_{deep}}$  cells are mapped to the same  $\mathcal{C}$  cell  $(1, 3)$ .

Therefore, given any pair of coordinates, it is possible to find the associated  $\mathcal{C}$  cell by first computing the associated  $\mathcal{C}^{l_{deep}}$  cell index, namely a pair  $(i, j)$ , and then performing a lookup in  $z_{map}$ . Both operations have constant complexity, even though we have to mention that the performance related to the lookups in  $z_{map}$  heavily depends on the ability to exploit the GPUs caching capabilities. Indeed,  $z_{map}$  may have a relevant size - depending on  $l_{deep}$ . In light of this, it is important the *memory layout* of  $z_{map}$  to enhance data locality.

As regards the *memory layout* of the bidimensional array  $z_{map}$ , instead of using the typical row-major order memory layout, we access it according to the Morton code obtained from index pairs  $(i, j)$  used to access the array. Since all objects and queries are first associated with the Morton code of the cell  $(i, j)$  in  $\mathcal{C}^{l_{deep}}$  which contains them, and then are *sorted* by this code, during the indexing operation described below we access  $z_{map}$  by exploiting temporal and spatial locality. This is because when we scan objects and queries that are memorized nearby, we also access nearby elements in  $z_{map}$ .

The initialization of  $z_{map}$  is performed entirely on GPU (function *GPUbuildZMap*, line 13 in Algorithm 1), by assigning each  $\mathcal{C}$  cell (quadtree leaf) to a GPU streaming multiprocessor, which in turn initializes the interval of cells (elements of the lookup table) in  $\mathcal{C}^{l_{deep}}$  contained by the  $\mathcal{C}$  cell assigned.

The complexity of *GPUbuildZMap* is  $O(|\mathcal{C}| + |\mathcal{C}^{l_{deep}}|)$  and, in practical terms, negligible.

(2,10)	(2,11)	(1,3)	
(2,8)	(2,9)		
(1,0)		(2,6)	(2,7)
		(2,4)	(2,5)

(2,10)	(2,11)	(1,3)	(1,3)
(2,8)	(2,9)	(1,3)	(1,3)
(1,0)	(1,0)	(2,6)	(2,7)
(1,0)	(1,0)	(2,4)	(2,5)

Figure 7: Example of the mapping established by  $z_{map}$  between the quadtree-induced grid  $\mathcal{C}$  (left side) and the uniform grid  $\mathcal{C}^{l_{deep}}$  (right side) related to the quadtree deepest level.

**Indexing moving objects and queries (QUAD).** The goal of this phase is to map objects locations and queries to  $\mathcal{C}$  cells (quadtree leaves). Each object location is mapped to a single cell while each query can be potentially mapped to multiple cells (Definition 6).

To convert the position of all objects in  $P$  to cells identifier  $c$  of  $\mathcal{C}$ , their 2-dimensional coordinates are first mapped to grid coordinates  $(i, j)$  in the  $\mathcal{C}^{l_{deep}}$  grid, where  $l_{deep}$  is the deepest level of  $\mathcal{C}$ . Subsequently, the Morton codes identifying the cells of  $\mathcal{C}^{l_{deep}}$  are derived from  $(i, j)$ . Then, objects are *sorted* according to such Morton codes in order to exploit caching when subsequently accessing  $z_{map}$ , where  $z_{map}$  is used to retrieve the final quadtree cell identifier  $c = z_{map}[i, j]$ ,  $c \in \mathcal{C}$ , in which the objects fall. We remark that objects remain sorted after such mapping thanks to quadtrees structural properties: this will be exploited during the filtering and decoding phases (see Sections 5.4 and 5.5), since query processing happens at cell level.

To convert a range query, characterized by a rectangular region, we need to identify all the relevant cells in  $\mathcal{C}$ , i.e., all the cells that spatially intersect the query. This process entails to identify, for each query  $q$ , a set of *subqueries*, each corresponding to the spatial restriction of the rectangular region of  $q$  to a relevant cell in  $\mathcal{C}$ . More formally, we have  $g(q) = \{c_1, c_2, \dots, c_n\} \subseteq \mathcal{C}$  (see Definition 6), where  $q$  intersects or covers each  $c_i \in \mathcal{C}$ . We thus refer to each pair  $(q, c_i)$  as a *subquery* of  $q$ .

First, as in the objects case, queries are first associated with a  $\mathcal{C}^{l_{deep}}$  cell, namely their Morton codes, through their reference corner (in case part of their spatial extent falls outside the MBR  $\mathcal{G}$ , only the area in common with  $\mathcal{G}$  is considered), and then sorted accordingly to such codes in order to exploit caching when subsequently accessing  $z_{map}$ .

Then, to obtain the subqueries, we start by identifying all the  $\mathcal{C}^{l_{deep}}$  cells intersected by the query. We map each of these cells identified by a pair  $(i, j)$  to the corresponding  $c \in \mathcal{C}$  cell by exploiting  $z_{map}$ . Depending on the spatial distribution, it is very likely to have multiple cells of  $\mathcal{C}^{l_{deep}}$  that intersect the range query, and are thus mapped to the same  $\mathcal{C}$  cell. This behavior could create *duplicate* subqueries, i.e., the same query mapped multiple times to the same cell of  $\mathcal{C}$ . Figure 8 illustrates the problem and sketches our solution to avoid the presence of multiple subqueries mapped to the same  $\mathcal{C}$  cell. In the left picture of the figure, we can see how the original query  $q_1$  falls over multiple  $\mathcal{C}$  cells (specifically, 6 distinct cells). Among these, we consider the intersection between  $q_1$  and the  $\mathcal{C}$  cell  $(1, 3)$  (yellow area). In the right picture, which illustrates the uniform grid  $\mathcal{C}^{l_{deep}}$  associated with  $\mathcal{C}$  through  $z_{map}$ , we can note that there are multiple cells on the right-upper part of  $q_1$  that map to cell  $(1, 3)$  in  $\mathcal{C}$  (specifically, 4 distinct cells of  $\mathcal{C}^{l_{deep}}$ ). Therefore,  $q_1$  would yield 4 subqueries that map to the same  $\mathcal{C}$  cell  $(1, 3)$ . To avoid duplicates, we always select the subquery having the minimal grid coordinates (highlighted in green).

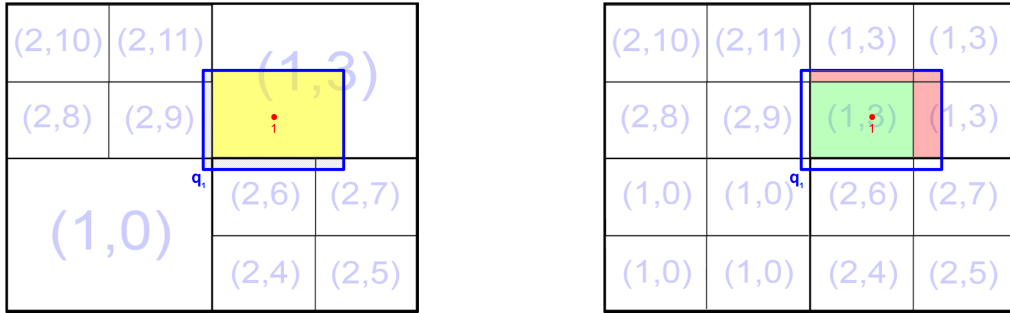


Figure 8: Query indexing example with QUAD.

The queries are indexed, similarly to UG ( $UG_{Baseline}$ ), in two separate phases. During the first phase the amount of subqueries per each original query is determined. In order to determine the memory location where each subquery will be written, an exclusive prefix sum is performed over the vector containing the amounts of subqueries per query. Then, in the second phase, subqueries are actually written using the information computed during the first phase, and classified according to the *intersecting/covering* dichotomy.

The overall complexity of the indexing phase can be expressed in the following terms:

- for what is related to the sorting operations needed to optimize the accesses in  $z_{map}$ , we have  $O(|P| + |Q| + b \cdot (|P| + |Q|)) \approx O(2(|P| + |Q|))$ :  $|P|$  and  $|Q|$  are due to  $l_{deep}$  Morton codes

computations while  $b \cdot (|P| + |Q|)$  relates to the actual sorting performed, by means of Radix Sort, over  $P$  and  $Q$ .

- for what is related to the subsequent operations, we have  $O(|P| + O(2|Q| \cdot |\mathcal{C}^{l_{deep}}| + |Q| + 2|\hat{Q}|))$ :  $|P|$  relates to the lookups in  $z_{map}$ ,  $2|Q| \cdot |\mathcal{C}^{l_{deep}}|$  relates to the query indexing which happens in two separate phases ( $|\mathcal{C}^{l_{deep}}|$  is due to the amount of  $\mathcal{C}^{l_{deep}}$  cells spanned by an original query in the *worst* case) and  $2|\hat{Q}|$  relates to the subqueries written during the second phase (lookups in  $z_{map}$  are included in the complexity), noting that  $|\hat{Q}| = |Q| \cdot |\mathcal{C}^{l_{deep}}|$  in the worst case.

In light of these considerations, the amount of subqueries to be checked during indexing may be relevant, therefore we remark the importance of exploiting caching when accessing  $z_{map}$ .

**Sorting (QUAD).** Once subqueries are mapped to  $\mathcal{C}$  cells, we sort the associated augmented tuples to store them in contiguous memory locations. The reason of this phase is analogous to the sorting carried out in the UG and UG<sub>Baseline</sub> cases.

Note that, unlike the UG and UG<sub>Baseline</sub> cases, for QUAD we do not need to re-sort the moving objects. The sorting done during the indexing phase, according to the cell identifiers of  $\mathcal{C}^{l_{deep}}$  grid, is enough to guarantee locality during the following query processing phase, thanks to the quadtree structural properties.

As regards subqueries, we have to sort them since there is no guarantee about the order in which they are written in global memory during the indexing phase. Consequently, the structure of vectors associated with the subqueries tuples,  $\hat{Q}$ , is sorted on GPU by means of Radix sort according to the identifier associated with the cell<sup>4</sup>. Moreover, in order to support the optimizations discussed in Section 5.6, each identifier is augmented so to signal whether a subquery is either covering or intersecting. In this way, after the sorting operation, all intersecting subqueries are placed at the beginning of their structure of vectors.

Since the sorting algorithm is Radix Sort, the complexity of the sorting step is  $O(b \cdot (|\hat{Q}|)) \approx O(|\hat{Q}|)$ .

## 5.4 Filtering

The goal of the filtering phase is to compute range queries over object locations, and store the containment test outcomes (i.e., which object locations are contained in each query range) conveniently. Since, by definition, covering subqueries entirely cover the cell onto which they fall, the filtering phase can be actually limited to intersecting subqueries, delegating the processing of the former type to the optimization described in Section 5.6.

In this context we conveniently denote by  $\mathcal{C}_\alpha \subseteq \mathcal{C}$  the set of active cells, i.e., those cells containing at least one object.

Both QUAD and UG store the containment test outcomes in form of bitmaps (one per *active* cell), which will be decoded at a later stage in order to obtain a final compact representation of the positive containment test outcomes.

Filtering is performed in parallel: each active cell in  $\mathcal{C}_\alpha$  is assigned to a block of GPU threads to obtain a bitmap which refers to object locations and subqueries falling in the corresponding cell.

In the last part of this subsection we also detail the simplifications adopted by the UG<sub>Baseline</sub> filtering algorithm. In the experimental section we will use UG<sub>Baseline</sub> as a baseline to assess the benefits of using the bitmaps and an additional (decoding) phase needed to extract the final query results from these.

**Bitmap layouts.** Bitmaps are arranged in memory by using two different layouts across the following phases, since each layout better fit specific kinds of operations on GPU. For each active cell  $c \in \mathcal{C}_\alpha$ , the filtering phase initially compute a 2D bitmap  $B^c$  characterized by an interlaced column-wise layout (Figure 9.a), where each column  $q_i$  refers to a single query and each row  $b_j$  refers to a fixed block of  $w$  object locations inside the cell. The width of each column is  $w$  bits (here we assume  $w = 32$ ) and the content of a  $w$ -bit word corresponding to  $q_i$  and  $b_j$  indicates if the object locations associated with block  $b_j$  are contained in the extent of query  $q_i$ . Thus, a single bitmap element  $B_{n,m}$ , where  $n$  and  $m$  are the

<sup>4</sup>While in the UG and UG<sub>Baseline</sub> cases this identifier is represented by an integer storing grid coordinates, in the QUAD case it is a pair  $(l, z)$ , which is indeed conveniently stored as an integer.

row and column at bit level, represents the containment test outcome between the  $(m/w)$ -th query and the  $((n \cdot w) + (m \bmod w))$ -th object location.

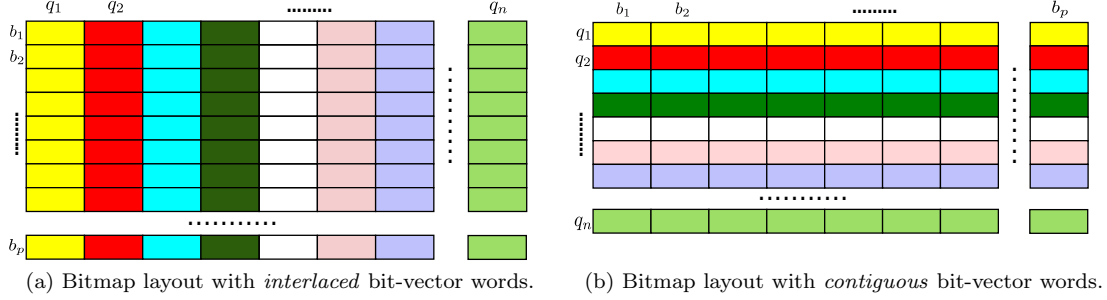


Figure 9: Bitmap layouts used during the processing.

This interlaced layout choice facilitates coalesced memory accesses when threads in the same warp are in charge of computing containment tests of a consecutive group of subqueries (see *Interlaced bitmap generation*). Indeed, these threads can write results to consecutive memory positions, taking advantage of coalesced memory accesses. Since threads in the same warp perform exactly the same operations, and each thread is in charge of writing a distinct 32-bit word, this solution eliminates the need of any synchronization mechanisms at thread block or global levels.

While the interlaced layout entails benefits when the bitmap is produced, it hinders the extraction of all the containment test outcomes referred to the same subquery. For this reason, after production, each interlaced bitmap is transposed word-wise to improve the memory throughput when the bitmaps are decoded to extract the final results. We will refer to this transformation as the *linearization* of interlaced bitmaps. Indeed, in the row-wise layout resulting from the transformation (Figure 9.b), single containment test outcomes are linearly indexed and bit-vectors associated with each subquery have their words arranged consecutively in memory, which favours subquery-wise read coalescing during the decoding phase. The linearization transformation can be expressed as a massively-parallel operation which is efficiently performed on GPU.

**Filtering – Interlaced bitmap generation.** During this stage we divide query result computation to exploit three different kind of parallelism allowed by GPUs.

*Block parallelism* allows to process independent tasks. Since we are considering subqueries, which are restricted to a specific index cell by definition, the computation of the results in different cells can proceed independently, producing distinct result bitmaps. Thus, active (non-empty) index cells  $c \in C_\alpha$  are assigned to distinct *blocks* of GPU threads.

Each block of GPU threads is executed asynchronously by the same *streaming multiprocessor* (SM). *Thread parallelism* allows for cooperation among threads in the same block. Each thread in a block is in charge of computing a distinct subquery that is present in the index cell assigned to the block. Since each bitmap is common for all the subqueries(threads) in a cell(block), the cooperation among threads is used to ensure coordination when writing out the containment test outcomes (0/1) in the bitmap.

Whenever possible it is strongly suggested to orchestrate the thread scheduling to hide memory access latency by having an amount of threads per thread block exceeding the amount of cores per single streaming multiprocessor. However, only subsets of threads can run in parallel at a given time. These subsets of  $sz_{warp}$ <sup>5</sup> synchronous and data parallel threads are called *warps*. Thanks to synchronous execution, *warp parallelism* allows to avoid synchronization operations. Furthermore, threads in the same warp benefit from coalesced memory accesses when they access consecutive (or identical) memory positions, so that several memory accesses are combined in a single transaction.

In our solution all the threads of a warp access the device memory in an optimal way: they read the same input data (object locations) synchronously (this exploits GPU caching), access them consecutively (subqueries, spatial locality, this exploits coalescing) and, thanks to the interlaced bitmap layout, write

<sup>5</sup>32 threads per warp in current GPUs.

simultaneously results ( $w$ -bits bitmap words) to consecutive memory locations (this entails coalesced memory access).

To better explain the latter point, we illustrate in Figure 10 the role of different threads in a thread block during the creation of an interlaced bitmap: each group of  $sZ_{warp}$  columns is collectively updated by a warp of threads in the thread block. Each column contains a set of 32 bit-wide words,  $b_1, \dots, b_p$ ,

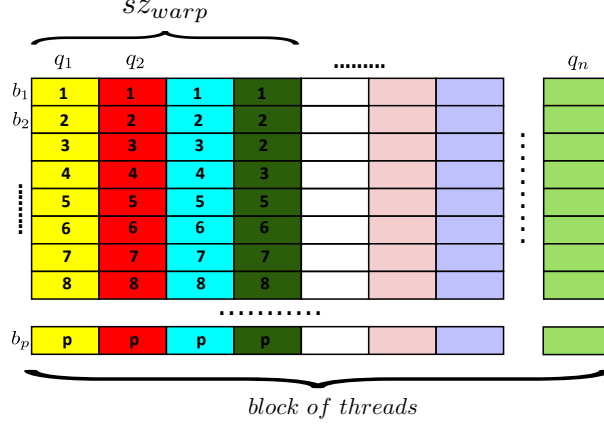


Figure 10: Collective creation of an interlaced bitmap by a block of GPU threads.

associated with a subquery  $q$ . The first words (the  $b_1$ 's) associated with the various subqueries, and computed by the warp threads, are stored simultaneously in memory. The same holds for the second block of words (the  $b_2$ 's), which is stored immediately after, and so on. The bitmap words updated simultaneously by the threads are stored consecutively thanks to the interlaced layout of the bitmap. This permits the writes to be *coalesced*.

---

**Algorithm 2:** QUAD and UG filtering phase.

---

```

1 begin
2    $numPoints \leftarrow 0$ 
3    $wordIndex \leftarrow 0$ 
4    $wordBitmap \leftarrow 0$ 
5   foreach  $c \in \mathcal{C}_\alpha$  parallelblock do
6     foreach  $q \in c$  parallelthread do
7       foreach  $p \in c$  do
8          $numPoints \leftarrow numPoints + 1$ 
9         if  $p \in q$  then
10           $setBit(wordBitmap, p)$ 
11         if  $numPoints \bmod 32 = 0$  then
12           $writeBitmap(wordBitmap, wordIndex, q)$ 
13           $wordBitmap \leftarrow 0$ 
14           $wordIndex \leftarrow wordIndex + 1$ 

```

---

The pseudocode in Algorithm 2 illustrates the main points of the interlaced bitmap generation. Distinct *blocks* of GPU threads process in parallel active index cells  $c \in \mathcal{C}_\alpha$  (line 5). Each thread in a block is in charge of computing the results of a distinct subquery present in  $c$  (line 6).

All threads read the same sequence of object positions and update a private *32 bit-wide* register that contains the bitwise information about the presence/absence of 32 distinct object locations in its own range query (line 10). When the threads in a warp have completed the update of the current word (i.e., they have finished to compute a block of 32 containment tests or they have computed all the blocks), all threads proceed by flushing the content of their private registers simultaneously to the global device

memory at the right memory displacement (line 12). The computation goes on until all the subqueries have been computed.

The execution of the inner loop (line 7) is scheduled by the GPU at warp level: it depends on resource availability and memory access latency, but threads in the same warp are granted to be synchronous. For example, *wordBitmap* will be completed simultaneously for all the threads in the same warp.

We finally note that all the threads of any warp access the device memory in an optimized way: they read the same input data synchronously (object positions, line 7) or consecutively (subqueries, line 6), thus always exploiting data spatial locality. Moreover, all threads write simultaneously words that are stored consecutively in memory, thus coalescing the writes and boosting the overall GPU global memory throughput (line 12).

**Filtering – Bitmap linearization.** The goal of this operation is to transform each bitmap from the interlaced column-wise layout to the linearized row-wise one, so that bitmaps can be more efficiently processed during the subsequent decoding phase (Section 5.5). This transformation is performed on GPU and is depicted in Figure 11. The Figure shows the work of a single warp composed of  $sz_{warp}$  threads. All the synchronous threads cooperate to transpose blocks of words, one block at a time. Each block is

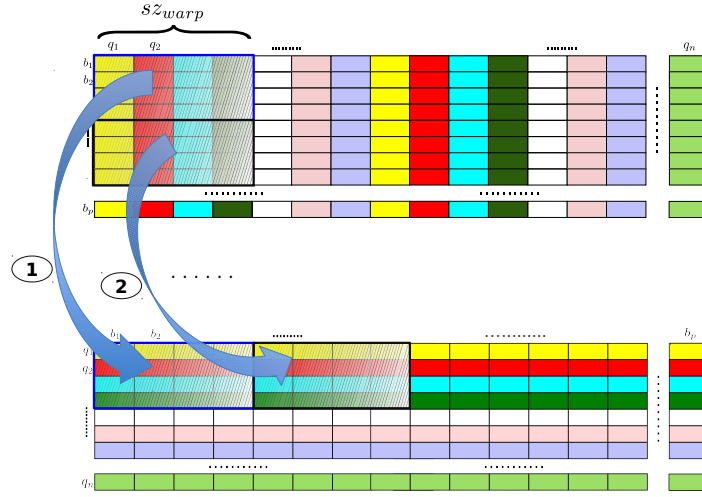


Figure 11: Collective linearization of a group of columns by a warp of GPU threads.

made up of  $sz_{warp} \times sz_{warp}$  words. The linearization of a block of words is carried out in two steps:

1. First, all the  $sz_{warp}$  threads read in parallel the bitmap words associated with the subqueries assigned to them (a row of the input block of words at a time). The threads proceed until the end of the block (for  $sz_{warp}$  rows of the block). While reading the words associated with subqueries, each thread incrementally prepares a linearised block of words in a temporary buffer stored in the fast shared memory available to each SM.
2. Once the input block has been read completely and linearized in the shared memory, the warp threads start their second step, where they move the linearized block to the device global memory. The only difference is that they collaborate by writing portions of the bitmaps associated with each subquery in parallel. First they write the first row of the linearised block in parallel, then the second, and so on. This, in turn, entails coalesced writes.

Even if the shared memory has limited size, the block-wise linearization does not saturate the shared memory thanks to the small size of the data that are linearized simultaneously.

**Filtering phase in  $UG_{Baseline}$ .** The  $UG_{Baseline}$  algorithm does not use bitmaps and computes the query results on the fly. Therefore, its filtering phase is simpler and does not require subsequent linearization and decoding phases. The pseudocode in Algorithm 3 illustrates this simpler strategy.



---

**Algorithm 3:**  $UG_{Baseline}$  filtering phase

---

```
1 begin
2   foreach  $c \in \mathcal{C}_\alpha$  parallelblock do
3      $shared\ resultBuffer \leftarrow \emptyset$ 
4     foreach  $q \in c$  parallelthread do
5       foreach  $p \in c$  do
6         if  $p \in q$  then
7            $appendResult(resultBuffer, q, p)$ 
8         if  $full(resultBuffer) = true$  then
9            $flush(resultBuffer)$ 
```

---

Each active cell is assigned to a single thread block (line 2). Query results (pairs of object locations and subqueries such that the object location is contained in the subquery range) are computed in parallel (line 6) and immediately appended to a buffer in shared memory (line 7) common to all the threads in the block. The threads access and update the buffer by means of *atomic operations* in order to guarantee its consistency. Once the buffer contains an amount of results greater than a fixed threshold (line 8), the threads synchronize and cooperatively flush its content out to global memory. In order to guarantee the consistency of the data written to the global memory a result counter, shared among all the thread blocks, is accessed and updated atomically.

**Filtering – Complexity.** The *interlaced bitmap generation* (Algorithm 2) represents the time dominant part of the filtering phase. The filtering complexity, determined by the amount of containment tests to be computed, is:

$$O\left(\sum_{c \in \mathcal{C}_\alpha} |\hat{Q}_I^c| \cdot |P^c|\right), \quad (3)$$

where  $\mathcal{C}_\alpha \subseteq \mathcal{C}$  is the set of active grid cells,  $P^c$  the set of object locations in  $c$  and  $\hat{Q}_I^c$  the intersecting subqueries associated with  $c$ .

In general we observe that decreasing the grid cells size yields a smaller number of containment tests, even if the number of intersecting subqueries to manage is larger. An arbitrary decrease, however, has negative side effects, such as the fragmentation of the intermediate results in a large number of small bitmaps, which in turn influences negatively the overall running time due to inefficient computational resource usage and scattered memory accesses. Moreover, an arbitrary cell size decrease may induce an intersecting subqueries increase rate eclipsing the decrease rate related to the average amount of objects per active cell, therefore raising the complexity at some point. In light of these considerations we argue there is a trade-off between decreasing the overall number of operations executed and optimizing parallelism and memory access costs.

The *bitmap linearization* has linear complexity with respect to the number of bitmaps words. Since each bit corresponds to an intersection test, the two subphases has the same complexity.

## 5.5 Bitmap decoding

The end product of the filtering phase of both **UG** and **QUAD** consists of a set of linearized bitmaps, one per active cell, containing both the positive and the negative containment test outcomes, related to object locations and subqueries associated with the active cells. The goal of the decoding phase is to process such bitmaps in order to extract the final query result set, i.e., the positive occurrences in the bitmaps.

Accessing the bitmaps content in order to extract the positive occurrences represents a memory and computationally intensive task which, thanks to the linearized layout, can be efficiently and conveniently parallelized on GPU.

The decoding phase, common to **UG** and **QUAD**, proceeds as follows: for each intersecting subquery a list of objects identifiers is generated (according to the problem definition in Section 2.5), where each

identifier represents a positive occurrence. Since each active cell (bitmap) represents a single task, the decoding operation can progress by decoding the tasks in chunks: this allows us to transmit to the CPU the information related to a previously decoded chunk of bitmaps while the GPU progresses by decoding the next chunk of unprocessed bitmaps. This also allows us to overlap computations carried on the GPU with I/O transfers from GPU to CPU. We highlight that the result lists related to intersecting subqueries originating from the same query have no results in common and preserve the identifier of the original query, so it is trivial to merge them to obtain the final result set.

The pseudocode in Algorithm 4 illustrates the GPU part of this strategy.

---

**Algorithm 4:** Decoding phase

---

```

1 begin
2   foreach bitmap parallelblock do
3     foreach  $q \in Q_{\text{bitmap}}$  parallelwarp do
4        $q_{id} \leftarrow \text{loadQueryID}(q)$ 
5       shared  $q_{bv} \leftarrow \text{loadQueryBitVector}(q, \text{bitmap})$ 
6        $\text{resultSet}_q \leftarrow \text{linearScan}(q_{bv})$ 
7       foreach  $r \in \text{resultSet}_q$  parallelthread do
8          $p \leftarrow \text{loadPoint}(r, \text{bitmap})$ 
9          $\text{writePID}(p, \text{bitmap})$ 
10       $\text{writeQueryDetails}(q_{id}, |\text{resultSet}_q|, \text{bitmap})$ 

```

---

Each bitmap refers to a specific index active cell and is decoded by a specific thread block (line 2). Each intersecting subquery referred by the bitmap is assigned to a *warp* (line 3), so that each warp is in charge of several subqueries. Since threads in the same warp are synchronous, the use of warp level granularity allows to safely avoid the use of synchronization mechanisms inside the loop.

Threads in the same warp transfer (line 5) the words composing the bit vector of the currently considered subquery from device memory to shared memory. Since consecutive threads read consecutive memory positions, this operation yields coalesced memory reads.

Once the transfer is over, each thread in the warp determines the subset of results it will write to global memory so that writes can be coalesced (i.e., the  $i$ -th warp thread will write the  $(i \bmod \text{warpSize})$ -th positive results). This is achieved by using the *linearScan* function (line 6). Here, each thread performs concurrently a linear scan over the query bit vector words in order to take note of the positive results it will write to global memory. This operation can be carried on efficiently thanks to the shared memories *broadcast* capability, which avoids bank conflicts between threads in a warp if they are all reading the same address. Moreover, each thread can store the information related to the positive results it has to write in its own private registers, since the decoding between lines 3 and 10 is scalable with respect to the subqueries bit vectors size (which is fixed for a given bitmap).

Once these information are determined, the warp threads finally perform a collective write of the subquery results, yielding coalesced writes (function *writePID*, line 9). Each warp also knows exactly where the results of each subquery has to be stored in global memory, since the overall amount of results per subquery can be determined during the filtering phase and stored in a vector (therefore, an exclusive prefix sum over the vector returns the correct memory location for each result).

**Decoding – Complexity.** Considered we have one bitmap for each active cell  $c$ , the overall decoding complexity (Algorithm 4) is:

$$O\left(\sum_{c \in \mathcal{C}_A} |\hat{Q}_I^c| \cdot |P^c| + \sum_{q' \in \hat{Q}_I^c} |R_{q'}|\right), \quad (4)$$

where  $R_q$  denotes the result set of a query  $q$ . The first term is due to the access for each cell  $c \in \mathcal{C}_A$  to the respective bitmap, each containing  $|\hat{Q}_I^c| \cdot |P^c|$  bits (number of intersecting subqueries times the number of objects in the cell), while the second term is due to the writes of the intersecting subqueries results. As we highlighted for filtering, the grid cells size indirectly affects the decoding complexity.

## 5.6 Optimizations

**Covering subqueries optimization – covering subqueries information notification.** UG, UG<sub>Baseline</sub> and QUAD take advantage of the covering subqueries (we denote their set by  $\hat{Q}_C$ ) in order to reduce the result set the GPU computes, thus saving a relevant amount of GPU computations during the filtering and decoding phases and I/O traffic between the GPU and the CPU during the decoding phase. This is achieved by notifying the CPU the covering subqueries data, together with the object locations enclosed in the  $\mathcal{C}$  cells, just after the end of the *sorting* phase.

**Covering subqueries optimization – covering subqueries result set expansion.** As soon as the data relevant for reconstructing the  $\hat{Q}_C$  result set is notified, the CPU can start its expansion. Indeed, for each  $q \in \hat{Q}_C$  we have to consider pairs  $(q, c \in \mathcal{C}_\alpha)$ , where each  $c$  represents the index of a cell entirely covered by  $q$ . After the final sorting step of the indexing phase, the lists of object locations associated with each cell of  $\mathcal{C}$  are sent to the host memory, so that the CPU can directly access them. Therefore, by looking at these lists, the CPU can immediately extract the result set related to  $q$ . This operation is performed by the CPU in background, between the end of the *sorting* phase and the end of the *decoding* phase of the GPU.

**Covering subqueries optimization – complexity.** The cost related to the covering subqueries result set expansion is  $O(\sum_{q \in \hat{Q}_C} |R_q|)$ , due to the scan of the list of object locations associated with the cell of each covering subquery. Since this task is carried out by the CPU and overlapped with the tasks performed on GPU, in practice it has very little or negligible impacts on the overall execution time.

**Task scheduling optimization.** A proper GPU task scheduling policy can substantially improve the overall execution time by reducing the inactivity time of the GPU streaming multiprocessors in presence of unbalanced workload distributions. In this context we define a single workload GPU *task* as the *set of computations* related to the intersecting subqueries falling inside a specific grid active cell, whereas the *computational weight* of a given task is the amount of containment tests associated with it, i.e., the product between the amount of intersecting subqueries and the amount of object locations falling inside the related active cell.

In general we can take into consideration three high-level GPU task scheduling strategies when assigning the workload tasks to the GPU streaming multiprocessors [20], namely, the static task list strategy (which is the default one used by CUDA), the task queue based strategy and the task stealing strategy.

Since the computational weight of each task is known a-priori once the sorting phase is performed, and that new sub-tasks cannot be created at run-time, we deem that the first strategy, together with a reordering of the task list according to the tasks computational weight, is the best one for the scenarios considered; indeed, this strategy has the effect of batching together the execution of tasks having similar computational costs at the negligible cost related to the need of accessing the task list atomically whenever an idling GPU streaming multiprocessor available to take in charge the first non-assigned task (atomic access is required to ensure a single execution for each task).

This optimization is used during the filtering and decoding phases when assigning tasks (active  $\mathcal{C}$  cells) to streaming multiprocessors.

## 6 Experimental setup

All the experiments are conducted on a PC equipped with an Intel Core i3 560 CPU, running at 3,2 GHz, with 4 GB RAM, and an Nvidia GTX 560 GPU with 1 GB of RAM coupled with CUDA 5.5. The OS is Ubuntu 12.04. We exploit a publicly available framework [21, 4] for both workload generation and testing. The framework comes with a number of sequential, CPU-based iterated spatial join algorithms. Among these, the *Synchronous Trasversal* algorithm (CPU-ST) is shown to be consistently the best [4] and thus we compare our GPU-based approaches with this algorithm.

As regards our GPU-based proposed solutions (QUAD and UG), we slightly modified the framework in order to offload the most time-consuming parallelizable tasks to the GPU, while delegating the others (mostly related to the GPU management) to the CPU.

We use three types of synthetic datasets: (i) *uniform datasets*, in which moving objects are distributed uniformly in the space; (ii) *gaussian datasets*, in which moving objects tend to gather around multiple *hotspots* by following a normal distribution. The skewness in the gaussian datasets depends on the number of hotspots: the more the hotspots are, the more the objects tend to be uniformly distributed; (iii) *network datasets*, in which moving objects are distributed uniformly over the edges of a bidirectional graph representing a road network. In our experiments we use the San Francisco road network, derived from TIGER/Line files. This kind of datasets are characterized by a mild skewness, due to the constraint on the positioning of the objects. All the datasets are created using the generator provided by the framework, which is partly derived from the Brinkoff generator [22].

In all tests we compute repeated range queries over 30 ticks. To model object movements the framework generates 30 instances of each dataset, one for each tick.

Table 1 summarizes the main parameters used to generate the datasets. The listed parameters apply to all the datasets, except for the amount of hotspots which is relevant for gaussian datasets only. The framework uses a generic spatial distance unit  $u$  (e.g., meters).

<i>Spatial region</i>	All tests occur in a squared spatial region with side length of 22500 $u$ .
<i>Amount of objects</i>	We vary the number of moving objects from 100K to 1500K. In some tests the number of moving objects is fixed and the exact amount is explicitly stated in their descriptions.
<i>Objects maximum speed</i>	In all tests the maximum speed of each object is fixed to 200 $u$ per tick ( $\Delta t$ ), where the objects are allowed to change their speed [4]. In general, changes in speed may slightly alter the objects distribution but do not change the distribution general properties.
<i>Query rate</i>	The percentage of objects that issue a range query during every tick is always set to 100%.
<i>Query size</i>	All queries in a test are squared and, depending on the experiment, they may be all equally sized or not. We vary the side length in the range [200 $u$ , 800 $u$ ]. The default value is 200 $u$ .
<i>Amount of ticks</i>	Whenever not specified, the default amount of ticks, corresponding to different snapshots of a dataset, is 30. Consecutive snapshots are expected to exhibit slight changes, according to the properties of the dataset spatial distribution.
<i>Query location</i>	All the queries are centered around the objects issuing them.
<i>Amount of hotspots</i>	Depending on the experiments goals and specificities, the amount of hotspots is varied in the [10, 150] range. Whenever not specified the default value used is 25.

Table 1: Data and workload generation parameters.

The decoded results are produced by the GPU in blocks, i.e., for each query/subquery a list of (positive) results is produced, whereas for CPU-ST the results are produced one by one. To avoid bias in the performance comparison, we thus force QUAD and UG to report the GPU-generated results to the framework in pairs, hence expanding the lists of objects belonging to the result set of each query.

## 7 Experimental evaluation

The experimental studies conducted for this work are introduced below, and are denoted by  $S1, \dots, S8$ :

- $S1$  We study how a *lock-free data structure*, like the bitmap proposed to encode the intermediate output of the range queries, entails considerable improvements over a baseline GPU algorithm that recurs to locks to assure the result buffer consistency.
- $S2$  We analyze the advantages coming from the *covering subquery optimization* in reducing computations (and related I/O traffic) performed on the GPU side.
- $S3$  We show how a proper GPU *task/block scheduling* can improve the UG and QUAD performances by reducing the workload unbalances deriving from skewed spatial distributions.

- S4* We study how the data distribution skewness influences the choice of the optimal grid coarseness, focusing on UG.
- S5* We study how QUAD is able to automatically adapt to the spatial data distribution of a dataset, even when the distribution is highly skewed.
- S6* We analyze the impact of various spatial distributions on the performance. To this regard we study mean and dispersion index related to the amount of objects per active cell achieved by UG and QUAD, and the relationships that these measures have with some important features that impact the performance of the system, such as the overall amount of subqueries and the proportion of covering/intersecting ones.
- S7* This study analyzes the sensitivity of the UG and QUAD performances with respect to datasets characterized by different spatial distribution properties, such as the amount of objects and the query area.
- S8* This final study analyzes how the main factors characterizing the datasets, such as the amount of objects, the query rate, the query area and the skewness, affect the system bandwidth  $\beta$  (as defined in Section 2.4).

It is worth remarking that in the final *S6...S8* studies we turn on all the optimizations devised for both UG and QUAD. In particular, a relevant amount of computations is avoided by distinguishing between covering and intersecting subqueries; we exploit the lock-free bitmaps to store the intersecting subqueries intermediate results; we heuristically balance the workload between the GPU SMs by reordering the tasks/blocks to be scheduled, from the heaviest to the most lightweight; finally, we always adopt the best possible grid coarseness for UG, given any dataset, by means of an oracle, although this strategy cannot be adopted in practical settings since it requires additional work to profile the UG performance for the various datasets.

## 7.1 Analysis on the benefits coming from the usage of bitmaps (S1)

In this study we evaluate the benefits coming from the usage of a lock-free data structure such as the bitmaps proposed. We focus on the filtering and decoding steps, since these are the only phases during which the bitmaps are utilized in order to significantly improve the performances. We limit this comparison to UG and UG<sub>Baseline</sub>, since QUAD filtering and decoding phases are similar to UG ones, and the benefits over a naive lock-based technique are thus analogous.

We briefly remember that in UG<sub>Baseline</sub> the query result set is computed and transmitted to the CPU counterpart on the fly, during the filtering step, thus avoiding the need of a subsequent decoding phase. Moreover, since the results of different queries can be interleaved in the output stream, each result is represented as a pair (*query:point*).

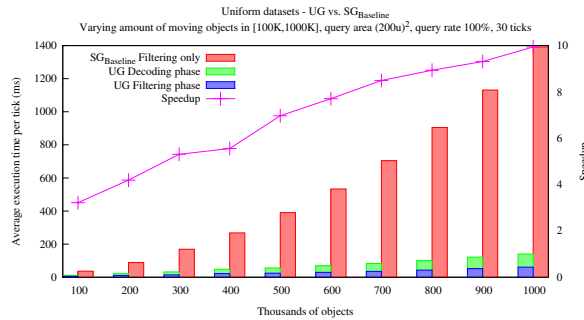


Figure 12: UG vs. UG<sub>Baseline</sub> time analysis for uniform datasets by varying the number of objects in [100K,1000K]. The histograms of the UG filtering and decoding phases are time-stacked for clarity purposes.

In Figure 12 we can see how UG outperforms UG<sub>Baseline</sub>, even when considering modest workloads. For our purposes, it is enough to compare UG and UG<sub>Baseline</sub> on a uniform spatial distribution of

object/queries. Note the line representing the speedup obtained by UG over UG<sub>Baseline</sub> per single experiment: the UG<sub>Baseline</sub> performance gets worse when the number of objects (and thus the output size) increases. This indicates that the main bottleneck of UG<sub>Baseline</sub> is the synchronization mechanisms adopted, which affect negatively the performance when the amount of results increases.

## 7.2 Covering subqueries optimization (S2)

The overall goal of this study is to analyze the benefits coming from the covering subqueries optimization described in Section 5.6. We briefly remember that this technique aims to speed up the query processing in three ways: first, by reducing the overall amount of containment tests performed by the GPU; second, by reducing the amount of results determined at the GPU side, and thus the amount of data the GPU has to send back to the CPU once the filtering and the decoding phases are over; third, by leaving the CPU in charge of expanding the covering subqueries result sets, during the same time that the GPU processes the intersecting subqueries.

We focus on QUAD, given its more advanced spatial indexing and considered that, in terms of query covering management, UG carries out the same operations per each subquery covering an index cell. We compare two different versions of QUAD: the former, denoted by **Covering ON**, is the version where the covering subquery optimization is exploited. The latter, denoted by **Covering OFF**, does not exploit the knowledge about the covering queries, thus considering all the subqueries as intersecting.

In our experiments we want to independently focus on two key parameters: spatial distribution *skewness* and *query area*. The skewness consistently influences (i) the *ratio* between covering and intersecting subqueries, and (ii) the *weight* of the covering subqueries in terms of the percentage of generated results. As regards (i), the more the objects tend to gather in specific places, the more QUAD refines the grid in those areas, in turn increasing the aforementioned ratio. As regards (ii), the smaller the average size of the QUAD cells is, the higher the probability that a subquery area completely covers a grid cell. Note that QUAD materializes dynamically the index cells, and generate smaller cells in correspondence to the spatial regions with higher object density. On these small and highly populated cells the ratio of covering subqueries to intersecting ones gets larger. This in turn increases the overall amount of results obtained from the covering subqueries, with positive returns on the performance.

Query area is another important factor, because it directly influences the ratio between covering and intersecting subqueries. Hence, we want to analyze how much this parameter influences the performances, aside from the skewness.

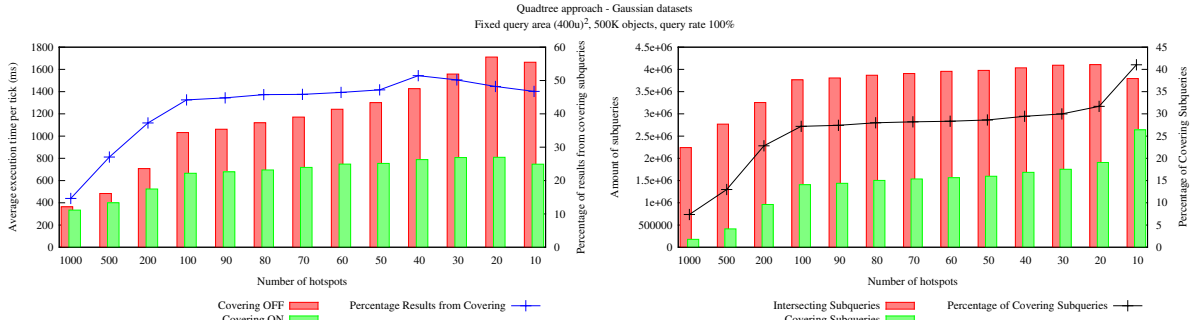


Figure 13: Gaussian datasets, 500K objects, query area  $(400u)^2$ , varying hotspots in  $[50,1000]$ , query rate 100%, Covering ON vs OFF.

In the first batch of experiments we vary the skewness of the object spatial distribution in a set of gaussian datasets, by changing the number of hotspots in the interval  $[10,1000]$ , while the amount of objects and the query area are kept fixed at 500K and  $(400u)^2$ , respectively. The left plot of Figure 13 shows how the exploitation of the covering subqueries greatly reduces the overall execution time, in particular when the skewness increases by reducing the amount of hotspots. The more the skewness is, the more the percentage of results coming from the covering subqueries is, thus increasing the performance gap between the two versions when the skewness gets larger. Finally, the right plot of Figure 13 gives a further explanation of the observed behaviour, and shows that the skewness is directly proportional to (or equivalently, the number of hotspots is inversely proportional to) the covering/intersecting ratio.



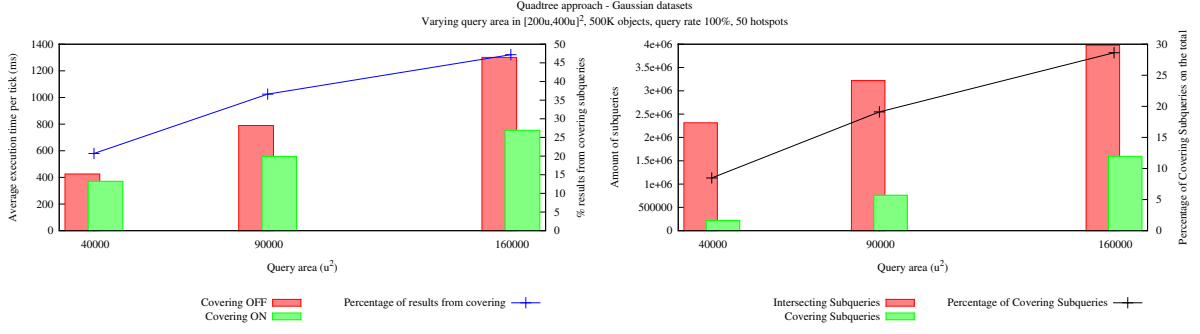


Figure 14: Gaussian datasets with 50 hotspots, 500K objects, query area varied in  $[(200u)^2, (400u)^2]$ , query rate 100%, Covering ON vs. OFF.

In the second batch of experiments we focus on a gaussian dataset having a fixed amount of 50 hotspots, thus characterized by a moderately skewed distribution. We vary the query area in the  $[(200u)^2, (400u)^2]$  range, while the amount of objects is kept fixed at 500K. Figure 14 shows that, when the query area is increased, the gap between the two versions of QUAD gets larger as well, while the covering/intersecting subqueries ratio strictly follows the trend.

In conclusion, we argue that the percentage of results coming from the covering subqueries determines the extent of the advantages possibly coming from this optimization. This figure is essentially determined by the skewness characterizing the spatial object distribution, while the query area amplifies or reduces this phenomenon by changing the query results redistribution ratio between the covering and the intersecting subqueries.

### 7.3 Task scheduling policy (S3)

In this section we investigate how the task scheduling optimization described in Section 5.6 can substantially improve the overall execution time of UG and QUAD by redistributing more evenly the workload among the GPU streaming multiprocessors. Besides analyzing the execution times in this study we also collect and study profiling data concerning the containment tests actually carried on by every streaming multiprocessor.

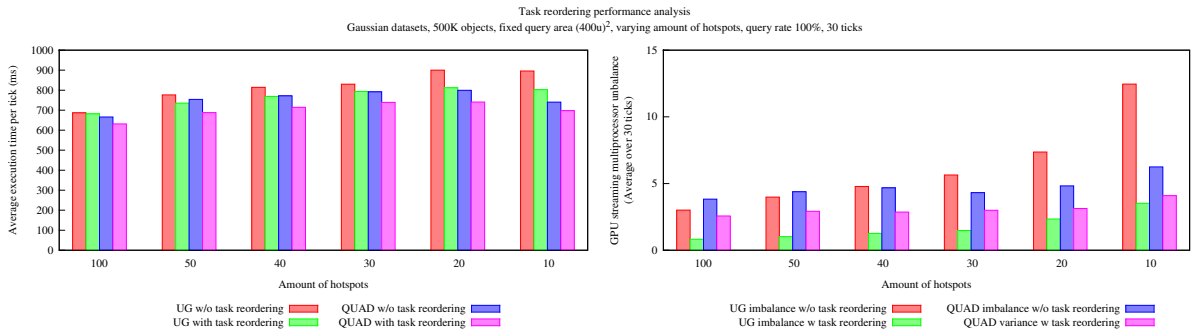


Figure 15: Analysis on the performances and workload redistribution among the GPU streaming multiprocessor with and without the static task list reordering - gaussian datasets, 500K objects, query area  $(400u)^2$ , query rate 100%, varying amount of hotspots. The left plot refers to the execution times observed while the right one refers to the profiling data collected during the filtering phase (decoding phase data is analogous).

The left plot in Figure 15 shows that the reordering always reduces the execution times of UG and QUAD. Note that in the case of UG the reordering entails higher performance improvements when the skewness gets large, while QUAD improvements are always moderate due to the ability of its underlying spatial indexing to dynamically produce tasks/blocks of similar weights.

We study in depth this behaviour by profiling the execution of the GPU SMs. In particular, we collect the per-tick amount of containment tests performed by each SM, and check whether the observed performance trends are reflected in workload unbalances among the SMs. In this context we define the *SM imbalance* measure during a single time tick as the *relative difference* between the highest amount of containment tests performed by a single GPU streaming multiprocessor with the lowest amount performed by another SM. Then, we compute the average of this measure across the ticks in order to characterize the average workload unbalance. From the right plot in Figure 15, we see how the trend of the *SM imbalance* follows the trend of the execution time, observed in the left plot of the same figure for both QUAD and UG. Hereinafter, all the experiments will be conducted by using the task list reordering optimization.

#### 7.4 Data skewness and optimal grid coarseness for UG (S4)

The following set of experiments aims to show that the best coarseness used for the uniform grid onto which the UG spatial indexing relies depends on the specificities of the spatial distribution characterizing the objects at each tick. We therefore aim to show how it is not possible to find a unique optimal MBR *split factor* (i.e., the number of columns/rows in which the MBR is decomposed) that holds for all the datasets. Even more, we show how each pipeline phase has its own optimal MBR split factor given a single dataset.

We first focus on a gaussian dataset characterized by a mild skewness (150 hotspots), and study how the UG performance changes (during a single time tick) by varying the split factor. We decompose the overall execution time in three macro phases, namely the *indexing*, *filtering*, and *decoding* phases, where the former includes the *index creation* and the *object/query indexing* phases (Sections 5.2 and 5.3). Figure

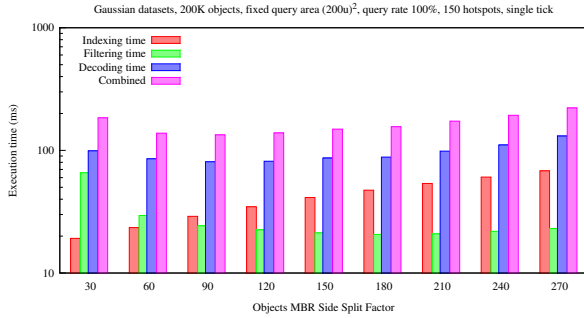


Figure 16: Gaussian dataset, 200K objects, query area  $(400u)^2$ , query rate 100%, 150 hotspots. The optimal value is equal to 110. Logscale on the y-axis is conveniently used to magnify small differences in the filtering execution times.

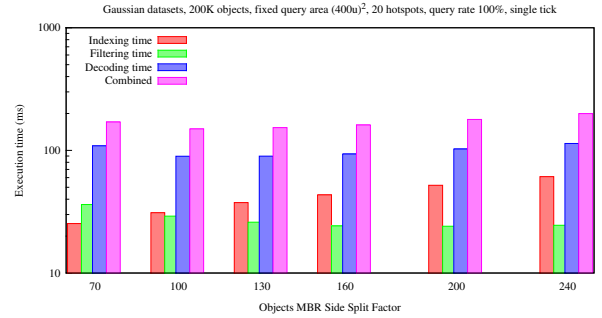


Figure 17: Gaussian dataset, 200K objects, query area  $(400u)^2$ , query rate 100%, 20 hotspots. The optimal value is equal to 95. Logscale on the y-axis is conveniently used to magnify small differences in the filtering execution times.

16 shows how the *indexing* time gets larger when we increase the MBR side split factor, as expected according to the costs described in Sections 5.2 and 5.3, due to the increase in the amount of subqueries created. As for the *filtering* phase, we see how the execution times trend exhibit a minimum. In general, too small split factors imply very large cells, few or none covering subqueries and potentially large workload unbalances, depending on the skewness. On the other hand, when the split factor is too large too many subqueries may be created, as well as there could be many active cells with small amounts of objects: this may represent a serious pitfall for an efficient usage of the memory/computational resources of a GPU. The same reasonings hold for the *decoding* phase as well. The overall execution time (the *Combined* bar in the plots) has a minimum obtained by using an optimal split factor equal to 110.

We replicate the same set of experiments with a consistently skewed gaussian dataset (Figure 17). The trends observed in Figure 16 are confirmed, although the UG optimal split factor value is different (95) due to the different dataset characteristics. This confirms that datasets having different spatial properties require the materialization of grids having different spatial characteristics in order to achieve the best possible performance.

## 7.5 Data skewness and optimal cell size for QUAD (S5)

As already described in Section 5.3, in QUAD the size of the various cells is determined dynamically on the basis of data distribution and according to  $th_{quad}$ , a threshold determining whether a quadtree quadrant needs to be split at the next level according to the amounts of objects it contains at the time tick the quadtree is computed. Thus, we need to determine an optimal value for  $th_{quad}$  which hopefully does not change for datasets characterized by different object spatial distribution or query areas.

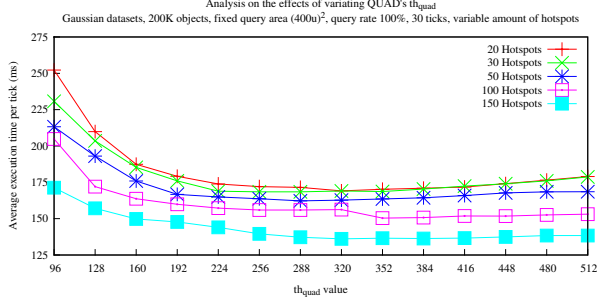


Figure 18: Performance analysis with different QUAD  $th_{quad}$  values when varying the skewness degree.

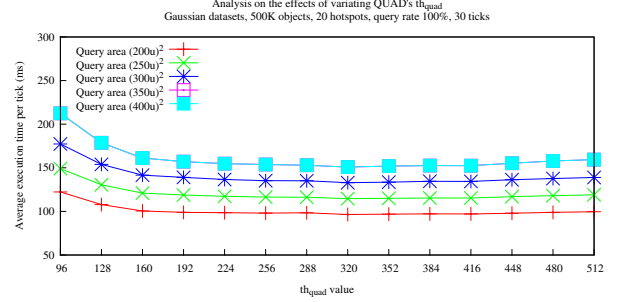


Figure 19: Performance analysis with different QUAD  $th_{quad}$  values when considering different query areas.

Figure 18 refers to a set of experiments in which, given a set of gaussian datasets with different amounts of hotspots,  $th_{quad}$  is varied in order to observe how QUAD behaves. The amount of objects characterizing each dataset is set to 200K, which allows the exploration of an extensive range of  $th_{quad}$  values: lower  $th_{quad}$  values increase the amount of resulting subqueries, which in turn increase the amount of GPU memory required for storing the subqueries. Figure 19 refers to a similar set of experiments in which the query area is varied among the datasets while the other characteristics are kept fixed.

In general we see how QUAD is resilient to dataset changes thanks to its low sensitivity with respect to  $th_{quad}$ , allowing an easy tuning of the system. Moreover, the search for an optimal  $th_{quad}$  is not so crucial, given the stability exhibited by QUAD for an ample interval of values. Increasing the query area has just the effect of increasing the execution times, while the trend remains the same for all the curves. Considering the results obtained above, in the experiments that follow we set  $th_{quad} = 384$ .

## 7.6 Impact of spatial distribution skewness on the performance (S6)

In this study we want to observe how UG and QUAD perform when varying the skewness degree by considering a set of gaussian datasets having different amounts of hotspots. In the experiments that follow we keep fixed the amount of objects (500K), the query area ( $400u^2$ ) and the query rate (100%), whereas we vary the amount of hotspots in the  $[10, 200]$  interval. For UG and QUAD we exploit all the optimizations, including the oracle used by UG (even though unusable in a practical setting).

Figure 20 shows that UG and QUAD have similar performances until the skewness becomes consistent, i.e., the amount of hotspots gets below 20. This is confirmed by the fact that QUAD is able to maintain stable and consistent speedups with respect to CPU-ST, even in presence of extremely skewed distributions, while UG slightly degrades.

We try to explain the observed performances in terms of the ability of UG and QUAD in redistributing the objects among the grid cells. To this end, we compute the *mean* and *variance* of the amount of objects in each grid active cell and the associated *dispersion index* ( $D = \sigma^2/\mu$ ) characterizing the distribution of the objects over the active cells. In general it is expected that, the finer a grid is, the lower the resulting mean and dispersion index are, although these figures are heavily influenced by the skewness characterizing the dataset. The mean and the dispersion indices obtained by UG and QUAD are shown in the top plot of Figure 21. UG always yields remarkably higher dispersion indices and lower means than the QUAD ones. The very low means observed for UG depend on the very fine uniform grid exploited, needed to avoid heavy populated cells which would entail very expensive tasks to be execute by a single GPU SM. Indeed, the ability of QUAD in properly redistributing workloads associated with objects living

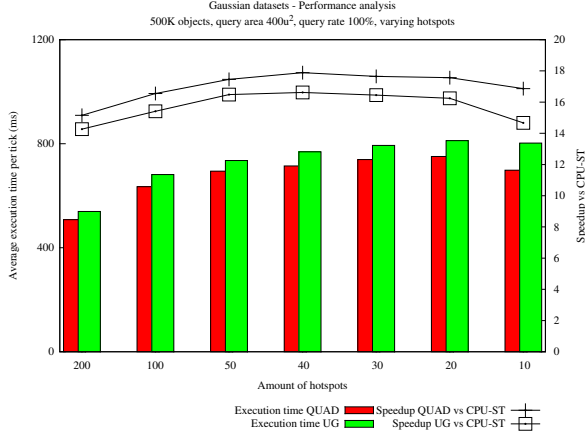


Figure 20: Gaussian datasets, 500K objects, query area  $(400u)^2$ , amount of hotspots varied in  $[10,200]$ , average running times per tick and speedup with respect to CPU-ST.

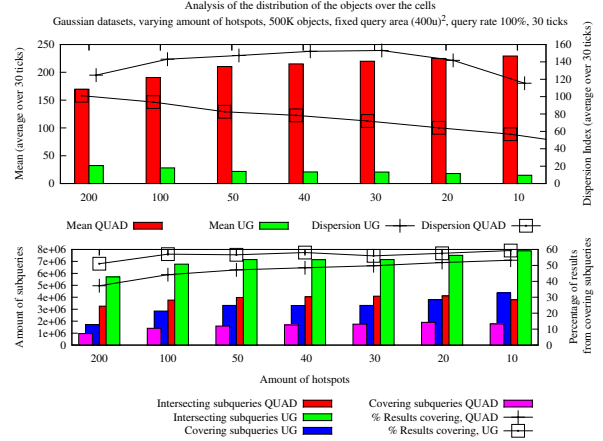


Figure 21: Gaussian datasets, 500K objects, query area  $(400u)^2$ , amount of hotspots varied in  $[10,200]$ , mean and dispersion index over the grid active cells.

in densely populated regions is confirmed by the overall amounts of (intersecting/covering) subqueries produced (Figure 21, bottom graph) - amounts which are remarkably lower than the ones obtained by UG. For example, the amount of subqueries obtained when analyzing a very skewed dataset (such as the gaussian one with 10 hotspots) is about 12 millions for UG, and approximatively half for QUAD. Even if the size of the UG cells is very small, and thus the probability that a subqueries “covers” a grid cell gets large, the same plot shows that the proportion of results coming from covering subqueries in QUAD are almost on a par with the one obtained by UG. Finally, the remarkable smaller count of subqueries allows QUAD to have lower GPU memory requirements than UG when generating and computing the subqueries.

## 7.7 Performance analysis for different spatial distributions, amount of objects, and query areas (S7)

In this study we analyze the performances of UG and QUAD with datasets characterized by different spatial distributions. In the experiments that follow we also vary the amount of objects and the query areas. For UG and QUAD we exploit all the optimizations. The goal is to show how QUAD is generally able to outperform UG, even if the latter relies on an expensive, and thus unfeasible, performance profiling (oracle) in order to select the best possible uniform grid coarseness for any dataset.

**Variable amount of moving objects.** In these experiments we exploit three types of datasets - uniform, gaussian and network-based - where we keep fixed the query rate and the query area at 100% and  $(200u)^2$ , respectively. For the gaussian datasets, the number of hotspots is fixed to 25. Figure 22 shows the execution times and the speedups versus CPU-ST for these three types of datasets when varying the amount of objects.

When uniform distributions are considered, UG and QUAD exhibit similar performances as expected. When gaussian datasets are considered, UG and QUAD exhibit stable and consistent performances, with QUAD performing noticeably better than UG. Finally, on network datasets UG and QUAD perform closely since these datasets are characterized by a very limited skeweness, with QUAD performing slightly better.

**Variable query area.** In this batch of experiments, whose results are shown in Figure 23, we vary the query area. All the queries are equally sized during a single experiment, while the amount of objects is fixed (700K for uniform, 500K for gaussian and network), as well as the query rate (100%) and the number of hotspots (25) for the gaussian datasets. With uniform distributions UG and QUAD again perform similarly. With gaussian distributions, UG and QUAD maintain consistent performances, even though the advantage of QUAD over UG still holds. Finally, with network datasets UG and QUAD are

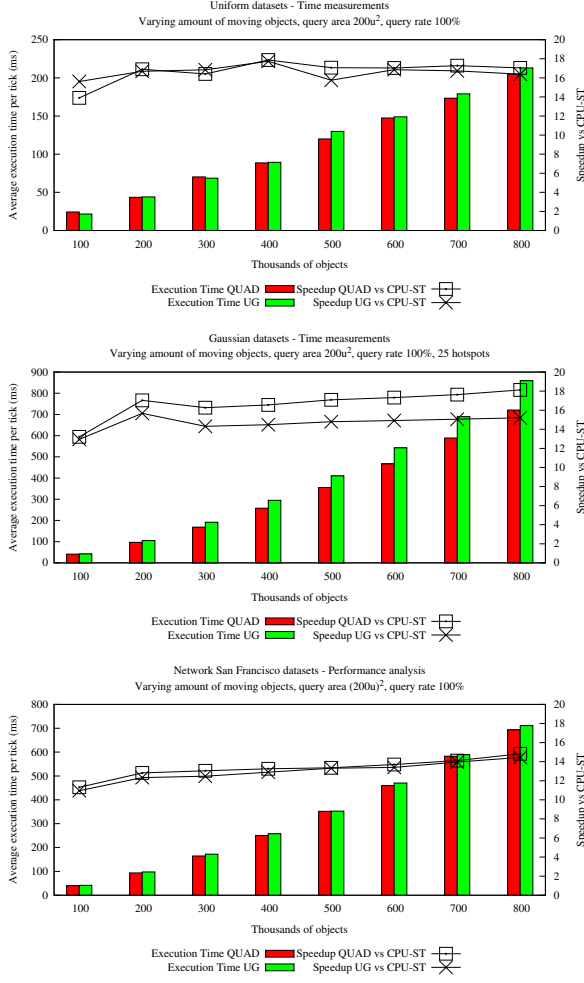


Figure 22: Varying the number of objects: average running time per tick and speedup versus CPU-ST. From top to bottom: uniform datasets, gaussian datasets with 25 hotspots, and San Francisco Network datasets.

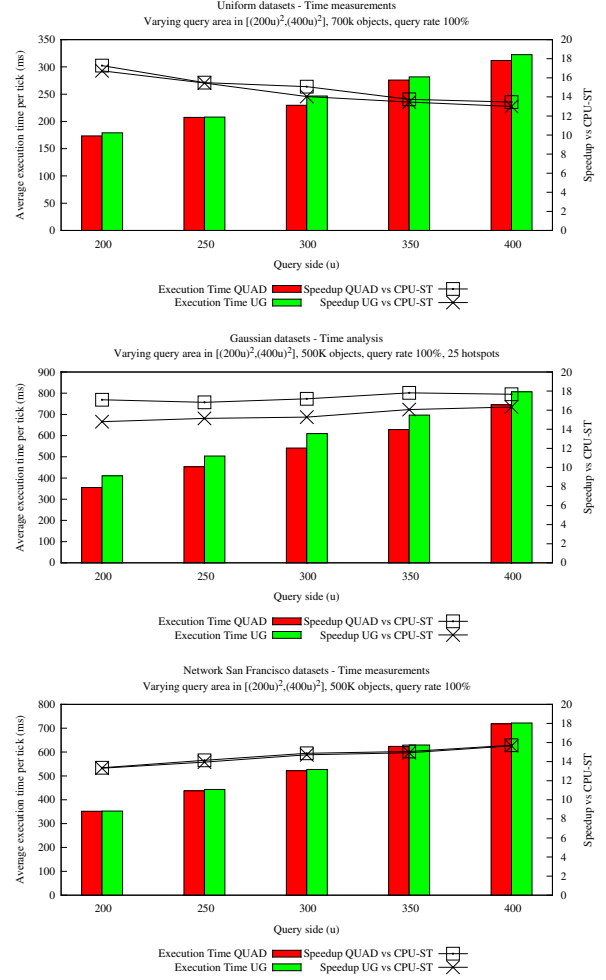


Figure 23: Varying the query area: average running time per tick and speedup versus CPU-ST. From top to bottom: uniform datasets, gaussian datasets with 25 hotspots, and San Francisco Network datasets.

almost on par, as already observed in the first batch of experiments, with a very slight advantage for QUAD.

**Variable amount of objects and variable query area.** In these experiments we consider different amounts of objects, each one issuing a query whose area is decided independently of the other objects and according to a uniform distribution in the  $[(200u)^2, (400u)^2]$  range.

For this experiments, we again consider uniform, gaussian (25 hotspots) and network datasets. In all the cases considered, the query rate is fixed at 100%.

The average execution times per tick measured in those tests are summarized in Figure 24. We observe that for that in the first and last case the two algorithms perform likewise, whereas QUAD outperform UG in the second case. Thus, even when the same dataset contains queries having different and uniformly distributed area, we can observe the same trends we highlighted in Study S7 for Figure 22.

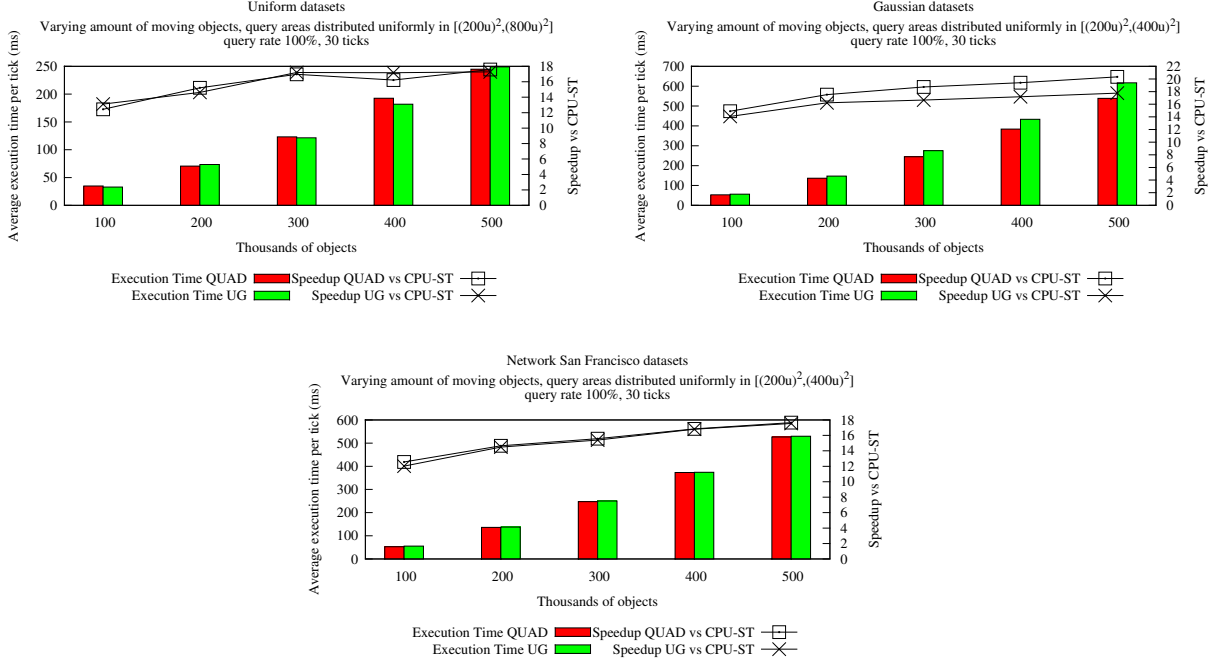


Figure 24: Variably sized queries: average running time per tick and speedup versus CPU-ST.

## 7.8 Bandwidth analysis (S8)

From lemma 2 in Section 2.4 we have that the system bandwidth  $\beta$ , expressed as the amount of queries processed per time unit (indeed, we use the second), is one of the crucial parameters in order to determine a suitable tick duration  $\Delta t$ , along with a given latency requirement  $\lambda$  and a maximum amount of queries which may occur during  $\Delta t$ ,  $Q_{max}$ . Since  $\lambda$  and  $Q_{max}$  are fixed, the crucial parameter becomes  $\beta$ .

Consequently, the goal of this study is to observe how the bandwidth  $\beta$  of a given system reacts to a set of dominant factors, such as the *amount* of moving objects, the *query rate* (i.e., the factor of moving objects issuing a query during a time unit), the *query area*, and the *skewness*. QUAD will be used to conduct all the experiments.

Figure 25 presents the results of the first batch of experiments, where we test the behaviour of  $\beta$  with respect to different amounts of objects and degrees of skewness. In order to conduct these experiments a set of gaussian datasets were considered. From the Figure we see how the system bandwidth decreases whenever the amount of objects or skewness degree (ranging from uniform-like distributions - 10000 hotspots - to moderately skewed ones - 25 hotspots) increase, due to an increase in the overall amount of containment tests and results that the underlying system must handle in the same time unit. We also observe how highly skewed datasets produce the most notable negative consequences on the performance, thus requiring particular care.

Figure 26 reports the results related to the second batch of experiments, where we analyze the behavior of  $\beta$  with respect to the query rate (we observe it corresponds to changing  $Q_{max}$ ) and the query area. To this end we consider a set of uniformly distributed datasets characterized by different query rates and query areas. We observe that increasing the query area decreases the system bandwidth, due to a quadratic increase in the amounts of containment tests and results the system must handle per time unit. As regards query rate, we see how the bandwidth increases whenever this parameter is increased. Even if this phenomenon may seem counter-intuitive at first, we observe that the action of increasing the query rate has the effect of increasing linearly (and not quadratically) the amount of containment tests and results produced. These increases, however, are compensated by an increased efficiency of the system. That is, the GPU resources are more utilized and thus better exploited, and this in turn increases the overall bandwidth. We note that we observed the same behavior for all of the spatial distribution we considered and for a wide range of choices of parameters.



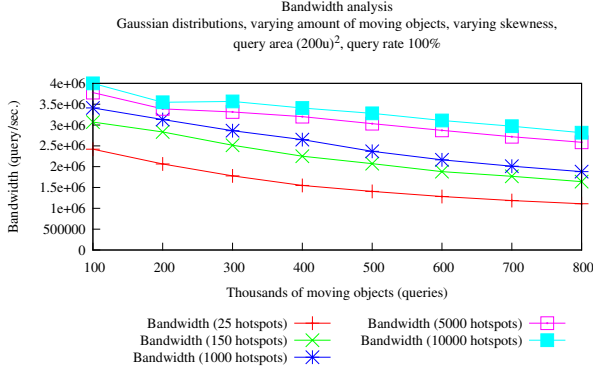


Figure 25: System bandwidth analysis when varying the amount of moving objects or the dataset skewness.

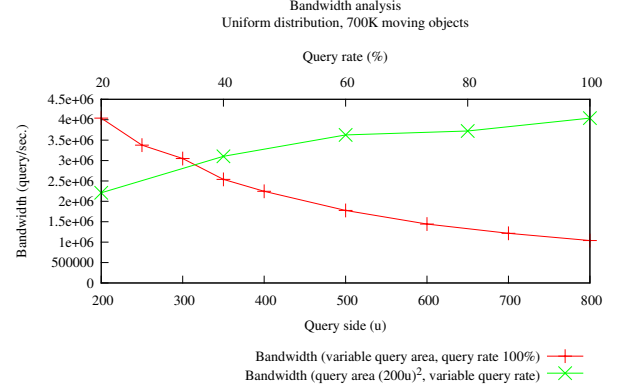


Figure 26: System bandwidth analysis when varying the query area or the query rate.

## 8 Related work

The idea of using the abundant and cheap computational power offered by GPUs in order to boost spatial joins computations dates back to the era when GPUs did not offer real general purpose computing capabilities and the use of OpenGL or DirectX APIs were needed in order to have access to their resources [23]. The potential of GPUs was clearly understood, but the scenarios, the problems, and the approaches considered at that time were quite different from those covered in this work.

As pointed out in an extensive review [4], the need for managing continuously incoming and evolving spatial data can be addressed by using simple, light-weight and, in many cases, throwaway data structures. However, it is crucial that data structures and algorithms contend effectively with skewed data and avoid redundant spatial joins and bad workload distributions as much as possible. In such review the authors claims Synchronous Traversal to be the top performer across several datasets. However, when considering its multi-threaded variant the speedup yielded by Synchronous Traversal (up to  $6\times$  with 12 cores) does not follow a linear behaviour as the amount of cores increases, due to inter-thread dependencies and challenges related to finding a proper way to partition the workload among the cores. Indeed, these are serious challenges which we try to tackle in the present work.

Recent studies [8, 13] show how uniform grid-based solutions are particularly attractive when managing continuously incoming and evolving spatial data in main-memory multi-core settings. Even if these works do not consider the architectural peculiarities and limitations of the GPUs, they nonetheless highlight how regular grids, in general, represent a natural basis for GPU parallelization strategies thanks to their structural regularity [16].

Other works consider the problem of building R-Trees (and possible derivations) from scratch [4], even recurring to hybrid approaches based on the combined use of CPU and GPU for range queries computation [24, 25]. While the goals of some of these works are different with respect from the ones of the present work, it is interesting to notice how solving certain problems is particularly recurrent and challenging when processing massive spatial data by using massively parallel architectures, i.e., (i) finding a solution able to distribute the workload in the most uniform way (depending also on the spatial data distribution), (ii) arranging spatial data by using proper GPU-friendly light-weight regular data structures which allow to use the GPUs features effectively, and (iii) exploiting spatial locality as much as possible.

In a recent work [26] in the context of collision detection in computer graphics, the the author focuses on extremely fast and efficient GPU-based construction and lookup algorithms for binary radix trees when performing real-time collision detection between 3D objects (thus addressing a similar problem with respect to the one addressed in this work). While the algorithms proposed are able to handle elegantly the skewness possibly characterizing the data, the work doesn't consider the problems of detecting and having to write out huge amounts of results in very short time intervals. The first problem increases remarkably the amount of lookups and traversals in the trees needed to compute a query, while the second problem entails serious issues mainly related to memory throughput maximization and how to

avoid memory access contention.

Previous works [16] already pointed out the advantages of using point-region quadrees for partitioning a low dimensional space when using the GPUs, thanks to the direct relationship between the quadrees structural properties and the Morton codes [14, Ch. 2][15]. Indeed, quadrees fit extremely well the GPUs architectural features, hence allowing to devise fast and efficient algorithms.

We are unaware of existing studies tackling the problem of repeatedly computing sets of range queries over continuously moving objects by means of an hybrid CPU/GPU approach. The most closely related work is focused on point-in-polygon joins [27]. This work considers scenarios characterized by massive amounts (possibly millions) of static entities, represented by points, and sets of polygons (in the order of few thousands) potentially covering the entities: the goal is to speed up the point-in-polygon tests by exploiting the computational power of GPUs using a novel approach stemming from the traditional *filtering and refinement* schema. This work is similar to the present work in that it exploits point-region quadrees in order to index the points, and thus improve the workload distribution when determining which point-in-polygon tests have to be computed. However, relevant differences separate the two works: (i) the entities are static, (ii) joins are computed between huge amounts of points and limited sets of polygons (instead of huge sets of range queries issued by the same entities) and (iii) polygons are indexed (through their bounding boxes) by means of a uniform grid and subsequently paired (for the final refinement phase) with sets of potentially overlapped points. This is in turn achieved by indexing each quadtree quadrant minimum bounding rectangle (enclosing the quadrant points) by means of the same uniform grid. In light of this, we deem that a comparison with our proposals would be not interesting, since the other work tackles different scenarios and consequently does not consider a set of relevant issues having far-reaching consequences, above all the issues related to the continuous management of huge sets of queries and results.

## 9 Conclusions

In this paper we present a novel method, relying on scalable grid-based spatial indices and on ad-hoc data structures, capable of computing massive amounts of repeated range queries over continuously moving objects. Since the range queries are repeatedly issued by the moving objects, also the queries continuously change their issuing location over time. The solution proposed is the first known to exploit GPUs in order to speed-up the query processing and, at the same time, effectively contend with skewed spatial distributions of objects and queries. To achieve these goals, we introduce an hybrid CPU/GPU query processing pipeline. We leverage bitmap-based intermediate data structures as well as quadtree-based spatial index, to exploit effectively the GPUs architectural features.

We extensively test our solution to study its sensitivity to parameters and data distribution. In such experiments we prove several arguments, above all that our solution (i) outperforms a baseline GPU approach thanks to the introduction of lock-free bitmaps used to handle intermediate query results and (ii) achieves a significant performance gain with respect to the best known CPU sequential competitor. We also show that (iii) our proposal is able to outperform an advanced GPU-based uniform grid-based solution, since solutions relying on such indices are unable to fully capture the data skewness - an ability which is needed in order to yield much more uniform workload distributions on GPUs.

As a future direction of research we plan to reuse, at least partly, the hybrid CPU/GPU processing pipeline introduced in this work in order to tackle the computation of repeated *k-nearest neighbours* queries issued by continuously moving objects. Indeed, this class of queries has vast amounts of potential applications, therefore speeding up significantly their processing would represent an important contribution.

## References

- [1] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.
- [2] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupati, Per Hammarlund, Ronak

- Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.
- [3] Claudio Silvestri, Francesco Lettich, Salvatore Orlando, and Christian S. Jensen. Gpu-based computing of repeated range queries over moving objects. In *Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '14, pages 640–647, Washington, DC, USA, 2014. IEEE Computer Society.
  - [4] Benjamin Sowell, Marcos Vaz Salles, Tuan Cao, Alan Demers, and Johannes Gehrke. An experimental analysis of iterated spatial joins in main memory. *Proc. VLDB Endow.*, 6(14):1882–1893, September 2013.
  - [5] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. *SIGMOD Rec.*, 22(2):237–246, June 1993.
  - [6] Dittrich J., Blunschi L., and Vaz Salles M.A. MOVIES: indexing moving objects by shooting index images. *Geoinformatica*, 15(4):727–767, 2011.
  - [7] Gray J. and Reuter A. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.
  - [8] Šidlauskas D., Šaltenis S., and Jensen C.S. Parallel main-memory indexing for moving-object query and update workloads. In *Proc. of ACM SIGMOD Conf.*, pages 37–48, 2012.
  - [9] Kornacker M., Mohan C., and Hellerstein J.M. Concurrency and recovery in generalized search trees. In *Proc. of ACM SIGMOD Conf.*, pages 62–72, 1997.
  - [10] Hong S. and Kim H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163, 2009.
  - [11] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 15–24. ACM, 2012.
  - [12] Hennessy J.L. and Patterson D.A. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.
  - [13] Šidlauskas D., Ross K.A., Jensen C.S., and Šaltenis S. Thread-level parallel indexing of update intensive moving-object workloads. In *Proc. of SSTD Conf.*, pages 186–204, 2011.
  - [14] Sariel Har-peled. *Geometric Approximation Algorithms*. American Mathematical Society, Boston, MA, USA, 2011.
  - [15] Rajeev Raman and David S Wise. Converting to and from dilated integers. *Computers, IEEE Transactions on*, 57(4):567–573, 2008.
  - [16] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009.
  - [17] Sengupta S., Harris M., Zhang Y., and Owens J.D. Scan primitives for GPU computing. In *Proc. of ACM SIGGRAPH Symposium on Graphics Hardware*, pages 97–106, 2007.
  - [18] Merrill D. and Grimshaw A.S. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(2):245–272, 2011.
  - [19] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.
  - [20] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 57–64. Eurographics Association, 2008.

- [21] Sowell B., Vaz Salles M., Cao T., Demers A., and Gehrke J. Indexing framework. <http://www.cs.cornell.edu/~sowell/indexing/>.
- [22] Thomas Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
- [23] Sun C., Agrawal D., and El Abbadi A. Hardware acceleration for spatial selections and joins. In *Proc. of ACM SIGMOD Conf.*, pages 455–466, 2003.
- [24] Luo L., Wong M.D.F., and Leong L. Parallel implementation of r-trees on the gpu. In *IEEE Conf. on Design Automation*, pages 353–358, 2012.
- [25] Yu B., Kim H., Choi W., and Kwon D. Parallel range query processing on R-tree with graphics processing unit. In *IEEE Conf. on Dependable, Autonomic and Secure Computing*, pages 1235–1242, 2011.
- [26] Tero Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *High Performance Graphics*, pages 33–37, 2012.
- [27] Zhang J. and You S. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In *Proc. of ACM SIGSPATIAL Intl. Wksp. on Analytics for Big Geospatial Data*, pages 23–32, 2012.